Python for Scientists

Release 205.09.22

Jonathan Leliaert and Toon Verstraelen

Sep 22, 2025

Contents

TODO: write some sort of overview here…

CHAPTER 1

Numerical Limitations

```
import decimal
import math
import struct

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

1.1 Approximations in scientific computation

1.1.1 Basic concepts

Absolute Error and Relative Error

Absolute Error = approximate value - true value

Relative Error = $\frac{\text{absolute error}}{\text{true value}}$

Another interpretation of relative error is that if an approximate value has a relative error of about 10^{-p}, then its decimal representation has about p correct **significant digits** (the leading nonzero digit and the $p-1$ following digits).

Precision and Accuracy

Precision: the number of digits with which a number is expressed.

Accuracy: the number of *correct* significant digits in an approximation of the desired quantity.

> **Example**
>
> 3.25260376469 is a very precise number but is not very accurate as an approximation for π. Computing a quantity using a given precision does not necessarily mean that the result will be accurate to that precision!

Truncation and Rounding error (Fout door de computer)

Truncation error: The difference between the true result and the result given by an algorithm using exact arithmetic. It is due to approximations such as truncating an infinite series or replacing derivatives by finite differences,…

Rounding error: The difference between the result produced by a given algorithm using exact arithmetic and the same algorithm, using finite-precision, rounded arithmetic.

1.2 Computer arithmetic

1.2.1 Floating-point number systems

In a digital computer, numbers are represented by a *floating-point* number system, which resembles *scientific notation* in which a number is expressed as a number of moderate size times an appropriate power of ten, e.g. 0.0007396 can be written as 7.396×10^{-4}. In this format, the decimal point moves, or *floats* as the power of 10 changes.

A floating-point number system is characterized by four integers:

symbol	name
β	Base or radix
p	Precision
$[L, U]$	Exponent range

Any floating-point number then has the form

$\pm \left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \cdots + \frac{d_{p-1}}{\beta^{p-1}} \right) \beta^E$ where d_i and E are integers such that $0 \leq d_i \leq \beta - 1$ and $L \leq E \leq U$.

A floating-point system is **normalized** if the leading digit d_0 always equals 1 (unless the number represented is zero).

This is advantageous because

- Each number has a unique representation.
- No digits are wasted on leading zeros, thereby maximizing precision.
- !! In a binary system ($\beta = 2$), the leading bit is always 1, and thus need not be stored, thereby gaining an additional bit of precision.

The two most important systems in use are the IEEE single precision (SP) and double precision (DP) standards with:

System	β	p	L	U
IEEE SP	2	24	-126	127
IEEE DP	2	53	-1022	1023

The single-precision binary floating-point exponent is encoded using an **offset-binary representation**, with the zero offset being 127

> **Example**
>
> Single precision numbers are stored in 4 bytes (or 32 bits), used as follows:
>
> - 1 sign bit
> - 8 bits for the exponent (ranging from -126 to 127)(the exponent sets the order of magnitude of the float)
> - 23 bits for the mantissa (the mantissa sets the actual precise value of the float)
>
> The exponent is calculated by taking -127 and then adding 2^n for every 1 in the 8 bits starting from the right. A

simple example 01011001 is the same as: $-127 + 1 + 8 + 16 + 64 = -38$

For instance the number 0.75 is stored as

0 01111110 10000000000000000000000 De eerste 0 betekend positief nummer

The float is calculated by multiplying a **number** (called the **mantissa** with a certain **sign** by 2 raised to a certain **power**.

The first digit defines the **sign** of the number: 1 meaning negative and 0 meaning positive. In this case we have a 0, so the float will be positive.

The following 8 digits define the **power**. The 8 digits refer to powers of 2. The first of them corresponds to 2^7, the second to 2^6 and so on till the eight digit referring to 2^0. To find the exponent we sum over these powers of 2 where the corresponding digit is 1. The value of this sum is than added to -127, the result is the exponent. In this case the exponent E is given by:

$$E = -127 + (2^1 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6) = -127 + 126 = -1$$

Note: *The smallest value for the exponent is L = -126 corresponding to the digits 00000001. The largest value for the exponent is U = 127 corresponding to the digits 11111110. The digits 00000000 and 11111111 are invalid combinations.*

The last 23 digits define the **mantissa** which is calculated by:

$\left(1 + \frac{d_1}{2} + \frac{d_2}{4} + ... \frac{d_{23}}{2^{23}}\right)$ De 1 staat er altijd automatisch, daarvoor moet nog niet gekeken worden naar de laatste 23 getallen

In the equation d_1 till d_{23} referred to the last 23 digits, being a 1 or a 0. In this case the following is found:

$$\left(1 + \frac{1}{2} + \frac{0}{4} + ... \frac{0}{2^{23}}\right) = 1.5$$

So in total we get:

$$\left(1 + \frac{1}{2} + \frac{0}{4} + ... \frac{0}{2^{23}}\right) 2^{(-127+126)} = 1.5 \cdot 2^{-1} = 0.75$$

If we would explicitly store d_0 (and allow for **denormalized** numbers with $d_0 = 0$), the representation would read

0 01111110 **1**10000000000000000000000 Hetzelfde nummer op een andere manier

Note that by adding the d_0 bit (in bold), we lost a bit of precision (d_{23}) because we only have 23 bits in total in the mantissa.

However, the same number could also be written as

0 01111111 **0**11000000000000000000000 , corresponding to $+\left(0 + \frac{1}{2} + \frac{1}{4} + ... \frac{0}{2^{23}}\right) 2^{(-127^{\text{offset}}+127)} = 0.75 \cdot 2^0 = 0.75$

This means that we've lost a bit of precision and gained nothing.

```
print(
    "python in itself only supports double precison floats,\n "
    "but numpy allows to use several different data types,\n "
    "including single precision floats\n"
)


def bit_rep(num):
    """Translate a number into its bit representation."""
```

(continues on next page)

(continued from previous page)

```
    return "".join(
        bin(c).replace("0b", "").rjust(8, "0") for c in struct.pack("!f", num)
    )


print("np.single(0.75) has bit representation of:", bit_rep(np.single(0.75)))
```

```
python in itself only supports double precison floats,
 but numpy allows to use several different data types,
 including single precision floats

np.single(0.75) has bit representation of: 00111111010000000000000000000000
```

> **Exercise**
>
> Convert this 32 bit single precision number to its decimal representation.
>
> 0 10000000 01101010000010011110011

1.2.2 Properties

A floating-point number is finite and discrete. The number of normalized floating-point numbers in a given system is $2(\beta-1)\beta^{p-1}(U-L+1)+1$

- 2 choices of sign
- $(\beta-1)$ choices for the leading digit of the mantissa $(=d_0)$
- β^{p-1} because there are β choices for each of the remaining $p-1$ digits of the mantissa
- $(U-L+1)$ possible values for the exponent (+1 because the boundaries of [L, U] are being counted)
- +1 because the number could be zero

The smallest positive normalized number (the **underflow level**): if all the bits in the mantissa part and all but the last bit in the exponent part are 0, the number equals

β^L

The largest number (the **overflow level**) equals

$\beta^{U+1}(1-\beta^{-p})$

> **Short derivation of the overflow level:**
>
> The largest possible value is the value where the upper limit U is reached and all values in the mantissa are equal to the maximum value of $(\beta-1)$
>
> the overflow level thus is $\beta^U(\beta-1)\cdot\sum_{k=0}^{p-1}\beta^{-k}$: for a IEEE single precision binary number this would look like
>
> 0 11111110 11111111111111111111111
>
> > If this was in ternary, all the ones would be two's, explaining the factor $\beta-1$.
>
> Note that not all values in the 8 bits of the exponent are 1. This form is reserved for `inf` and `NaN`.

Hence we find:

$$\beta^U(\beta-1)\cdot\sum_{k=0}^{p-1}\beta^{-k}=\beta^U(\beta-1)\cdot\frac{\sum_{k=0}^{p-1}\beta^k}{\beta^{p-1}}=\beta^U(\beta-1)\cdot\frac{1}{\beta^{p-1}}\cdot\frac{\beta^p-1}{\beta-1}=\beta^{U+1}\cdot(1-\beta^{-p})$$

Example

Let's look at two examples of how the overflow level is calculated.

First lets look at the situation where p = 24 and $\beta = 2$.

The overflow level is then:

$2^U\left(1+\frac{1}{2}+\frac{1}{4}+....\frac{1}{2^{23}}\right)=2^{U+1}\left(\frac{1}{2}+\frac{1}{4}+....\frac{1}{2^{24}}\right)=2^{U+1}\left(1-\frac{1}{2^{24}}\right)$

- **Note:** for $\beta = 2$ the d_i are always either 0 or 1 since $0 \leq d_i \leq \beta - 1$. The maximum value for the d_i thus is 1.

For the second example lets look at the situation where p = 50 and $\beta = 10$.

The overflow level is then:

$10^U\left(9+\frac{9}{10}+\frac{9}{100}+....\frac{9}{10^{49}}\right)=10^{U+1}\left(\frac{9}{10}+\frac{9}{100}+....\frac{9}{10^{50}}\right)=10^{U+1}\left(1-\frac{1}{10^{50}}\right)$

- **Note:** for $\beta = 10$ the maximum value for the d_i is 9.

As expected, for both cases the overflow level is given by: $\beta^{U+1}(1-\beta^{-p})$.

Example

Now, let's take a look at a *toy* floating point system with β=2, p=3, L=-1 and U=1.

This system supports 25 floating point-numbers:

$2(\beta-1)\beta^{p-1}(U-L+1)+1=2(2-1)2^{3-1}(1-(-1)+1)+1=25$

The largest number is $\beta^U(\beta-1)\cdot\sum_{k=0}^{p-1}\beta^{-k}=2^2\cdot 1(1-2^{-3})=3.5$

The smallest number is $\beta^L=2^{-1}=0.5$

Floating-point numbers are not uniformly distributed throughout their range, but are equally spaced only between successive powers of β.

A comparison of small systems is shown in the graph below. Note that, although these systems are extremely small, they are representative for all float-point systems in their property that they are unevenly spaced. Try larger values of p=5, p=8 to see the density grow.

```
def generate_custom_floats():
    # Custom floating-point format parameters
    sign_bits = ["0", "1"]  # 1 bit for sign
    exponent_bits_list = [
        "01",
        "10",
        "11",
    ]  # 2 bits for exponent ('00' reserved for zero)
    mantissa_bits_list = ["00", "01", "10", "11"]  # 2 bits for mantissa
```

(continues on next page)

(continued from previous page)

```
    bias = 2  # Exponent bias
    m = 2  # Number of mantissa bits (excluding implicit leading 1)
    float_values = []  # List to hold values
    bit_strings = []  # List to hold bit strings

    # Manually add zero (special case where exponent and mantissa are zero)
    zero_positive_bit_string = "0 00 00"
    zero_negative_bit_string = "1 00 00"
    zero_positive_value = 0.0
    zero_negative_value = -0.0
    float_values.append(zero_negative_value)
    bit_strings.append(zero_negative_bit_string)
    float_values.append(zero_positive_value)
    bit_strings.append(zero_positive_bit_string)

    for sign_bit in sign_bits:
        for exponent_bits in exponent_bits_list:
            exponent_value = int(exponent_bits, 2)
            E = exponent_value - bias
            for mantissa_bits in mantissa_bits_list:
                # Skip adding zero again
                if exponent_bits == "00" and mantissa_bits == "00":
                    continue
                mantissa_value = int(mantissa_bits, 2)
                mantissa = 1 + mantissa_value * (2**-m)
                value = (-1) ** int(sign_bit) * mantissa * (2**E)
                # Create bit string with spaces between sign, exponent, and␣
↪mantissa
                bit_string = f"{sign_bit} {exponent_bits} {mantissa_bits}"
                float_values.append(value)
                bit_strings.append(bit_string)

    # Sort the lists based on the floating-point values
    sorted_indices = sorted(range(len(float_values)), key=lambda i: float_
↪values[i])
    sorted_float_values = [float_values[i] for i in sorted_indices]
    sorted_bit_strings = [bit_strings[i] for i in sorted_indices]

    return sorted_float_values, sorted_bit_strings


def plot_custom_float_distribution():
    # Generate the floats and their corresponding bit strings
    float_values, bit_strings = generate_custom_floats()

    # Filter values and bit strings within the range -3.5 to 3.5 separately
    filtered_values = [v for v in float_values if -3.5 <= v <= 3.5]
    filtered_bits = [
        bit_strings[i] for i, v in enumerate(float_values) if -3.5 <= v <= 3.5
    ]

    # Print the sorted list with binary representations
    print("Sorted Floating-Point Numbers with Binary Representations:")
    for val, bits in zip(filtered_values, filtered_bits, strict=False):
```

(continues on next page)

(continued from previous page)

```
        print(f"{val:6.3f}   {bits}")

    # Plotting
    plt.close("custom_float")
    fig, ax = plt.subplots(figsize=(12, 3), num="custom_float")
    ax.set_title("Distribution of Custom Floating-Point Numbers")
    ax.set_yticks([])
    ax.spines["left"].set_visible(False)

    # Plot the floating-point numbers
    ax.plot(
        filtered_values,
        [0] * len(filtered_values),
        marker="|",
        linestyle="None",
        markersize=10,
    )

    # Add horizontal line (x-axis)
    ax.axhline(y=0, color="black", linewidth=0.5)

    # Display all numbers on the x-axis
    ax.set_xticks(
        filtered_values, [f"{v:.3f}" for v in filtered_values], rotation=90
    )


plot_custom_float_distribution()
```

```
Sorted Floating-Point Numbers with Binary Representations:
-3.500   1 11 11
-3.000   1 11 10
-2.500   1 11 01
-2.000   1 11 00
-1.750   1 10 11
-1.500   1 10 10
-1.250   1 10 01
-1.000   1 10 00
-0.875   1 01 11
-0.750   1 01 10
-0.625   1 01 01
-0.500   1 01 00
-0.000   1 00 00
 0.000   0 00 00
 0.500   0 01 00
 0.625   0 01 01
 0.750   0 01 10
 0.875   0 01 11
 1.000   0 10 00
 1.250   0 10 01
 1.500   0 10 10
 1.750   0 10 11
 2.000   0 11 00
 2.500   0 11 01
```

0.625=6·10^{−1}+2·10^{−2}+5·10^{−3}

0 = positief
01 = exponentveld (met bias = 1)
01 = mantisseveld (in 2 bits)

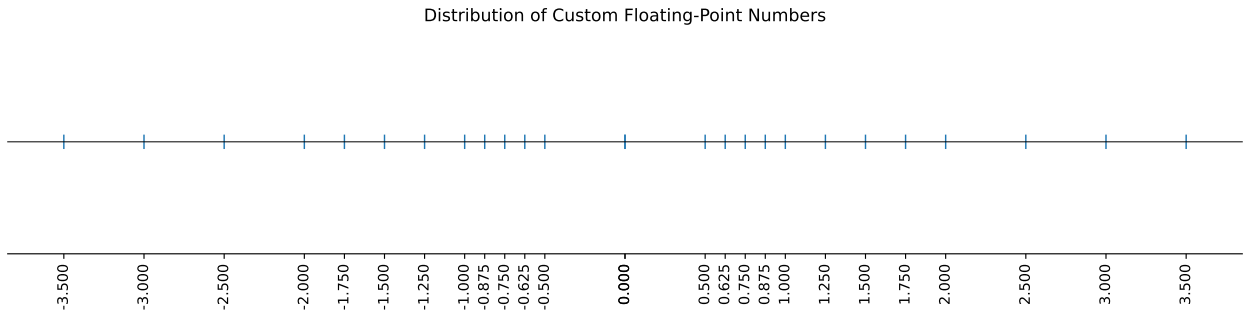
Methode: vermenigvuldig het getal met 2 en neem telkens het integer-gedeelte als bit.
1) 0.625 × 2 = 1.25 → eerste bit = 1, rest = 0.25
2) 0.25×2=0.5 → tweede bit = 0, rest = 0.5
3) 0.5×2=1.0 → derde bit = 1, rest = 0

(continues on next page)

Conclusie:
0.625_10 = 0.101_2

(continued from previous page)

```
 3.000   0 11 10
 3.500   0 11 11
```



Real numbers that are exactly representable in given floating-point system are called **machine numbers**. If a given number is not representable, it must be rounded to a "nearby" floating-point number. The error introduced by this approximation is called the **rounding error**. The most accurate and unbiased rule to round (and the de-facto standard today, also in the IEEE standards) is **round to nearest**, where a number is represented by its nearest floating-point number. In case of a tie, we use the number whose last stored digit is even. An alternative would be **chopped**: the expansion in $\frac{d_i}{\beta^i}$ is truncated after the (p-1)st digit, i.e. $\frac{d_{p-2}}{\beta^{p-2}}$.

The accuracy of a floating-point system is called the **machine precision**. Its value depends on the particular rounding rules that are being used. In case of rounding to nearest it equals to: $\epsilon_{\text{mach}} = \frac{1}{2}\beta^{1-p}$

> **Example**
>
> For the IEEE SP and DP systems, $\epsilon_{\text{mach}} = 2^{-24} \approx 10^{-7}$ and $\epsilon_{\text{mach}} = 2^{-53} \approx 10^{-16}$, respectively. These systems thus have about 7 and 16 decimal digits of precision.

Dit is waarom 0.3 niet gelijk is aan 0.1+0.1+0.1 (beiden geen machine number, worden beiden afgerond).

Although both values are small, ϵ_{mach} should not be confused with the underflow level.

Finally, there are two additional special values to indicate exceptional situations:

- `Inf`, which stand for **infinity**, which results e.g. from dividing a non-zero number by zero.
- `NaN`, which stands for **not a number**, and results from an undefined operations such as $\frac{0}{0}$.

```
print(
    "The largest double precision number is about 1.8e308, "
    "larger numbers become infinity."
)
print(1.797e308, 1.798e308)
print("\n")

print(
    "The smallest double precision number is about 4.95e-324, "
    "numbers that are smaller than half this value get rounded to zero."
)
print(4.95e-324, 2.4e-324)
print("\n")

print("Because 0.1 is stored as: ")
print("\t", decimal.Decimal(0.1))
print("and 0.3 is stored as ")
print("\t", decimal.Decimal(0.3))
```

(continues on next page)

(continued from previous page)

```
print(".1+.1+.1 does not equal .3")
print("As shown by testing .1+.1+.1 == .3, which gives:", 0.1 + 0.1 + 0.1 ==␣
↪0.3)
print("Instead, it equals", 0.1 + 0.1 + 0.1)
print("which differs from .3 by", 0.1 + 0.1 + 0.1 - 0.3, "\n")
```

```
The largest double precision number is about 1.8e308, larger numbers become␣
↪infinity.
1.797e+308 inf


The smallest double precision number is about 4.95e-324, numbers that are␣
↪smaller than half this value get rounded to zero.
5e-324 0.0


Because 0.1 is stored as:
         0.1000000000000000055511151231257827021181583404541015625
and 0.3 is stored as
         0.2999999999999999888977697537484345957636833190917968 75
.1+.1+.1 does not equal .3
As shown by testing .1+.1+.1 == .3, which gives: False
Instead, it equals 0.30000000000000004
which differs from .3 by 5.551115123125783e-17
```

1.2.3 Good Practices for computer arithmetic

Cancellation

- Avoid subtracting two almost identical numbers

Addition

- Avoid adding small and large numbers —> Is niet accuraat, je verliest informatie
- Perform a sequence of additions ordered from the smallest number to the largest

```
def example_cancellation():
    print("Example of cancellation error")
    x = 0.1234567891234567890
    y = 0.1234567891234567
    print(
        "The real value of x - y is 8.9e-17, "
        "however python returns a number which is about 7% smaller"
    )
    print(x - y, "\n")


example_cancellation()
```

```
Example of cancellation error
The real value of x - y is 8.9e-17, however python returns a number which is␣
↪about 7% smaller
8.326672684688674e-17
```

Consider the following sum of an alternating harmonic series

$$S = \sum_{k=1}^{N} (-1)^{k+1} \frac{1}{k} \approx \ln(2)$$

for large k.

```
# Function to sum in natural order with single precision
def alternating_harmonic_natural(N):
    total = np.float32(0.0)
    print("Summing alternating harmonic series in natural order (from 1 to N):
 ↪")
    for k in range(1, N + 1):
        term = np.float32((-1) ** (k + 1) / k)
        total += term
    true_value = np.float32(math.log(2))  # True value is ln(2)
    relative_error = (total - true_value) / true_value
    print(f"\tTotal sum: {total:.16f}")
    print(f"\tRelative error: {relative_error:.16e}\n")
    return total


# Function to sum in reverse order with single precision
def alternating_harmonic_reverse(N):
    total = np.float32(0.0)
    print("Summing alternating harmonic series in reverse order (from N to 1):
 ↪")
    for k in range(N, 0, -1):
        term = np.float32((-1) ** (k + 1) / k)
        total += term
    true_value = np.float32(math.log(2))  # True value is ln(2)
    relative_error = (total - true_value) / true_value
    print(f"\tTotal sum: {total:.16f}")
    print(f"\tRelative error: {relative_error:.16e}\n")
    return total


# Function to sum positive and negative terms separately in single precision
def alternating_harmonic_grouped(N):
    print("Summing positive and negative terms separately:")
    total_positive = np.float32(
        sum(np.float32(1.0 / k) for k in range(1, N + 1, 2))
    )  # Positive terms
    total_negative = np.float32(
        sum(np.float32(-1.0 / k) for k in range(2, N + 1, 2))
    )  # Negative terms
    total = total_positive + total_negative
    true_value = np.float32(math.log(2))  # True value is ln(2)
    relative_error = (total - true_value) / true_value
    print(f"\tTotal sum: {total:.16f}")
    print(f"\tRelative error: {relative_error:.16e}\n")
    return total


# Main block to execute all summation methods in single precision
N = 10000000
```

(continues on next page)

(continued from previous page)

```
print(
    f"Computing the alternating harmonic series up to N = {N} in single␣
 ↪precision\n"
)
print(f"The true value is ln(2) ≈ {np.float32(math.log(2)):.16f}\n")

# Compute and compare the results
sum_natural = alternating_harmonic_natural(N)
sum_reverse = alternating_harmonic_reverse(N)
sum_grouped = alternating_harmonic_grouped(N)
```

```
Computing the alternating harmonic series up to N = 10000000 in single␣
 ↪precision

The true value is ln(2) ≈ 0.6931471824645996

Summing alternating harmonic series in natural order (from 1 to N):
        Total sum: 0.6931374669075012
        Relative error: -1.4016585737408604e-05

Summing alternating harmonic series in reverse order (from N to 1):
        Total sum: 0.6931471228599548
        Relative error: -8.5991324283440917e-08

Summing positive and negative terms separately:
        Total sum: 0.2981586456298828
        Relative error: -5.6984800100326538e-01
```

Even relatively simple mathematical problems can exhibit numerical issues when computed using finite-precision arithmetic. A classic example is the **quadratic formula** used to find the roots of a quadratic equation.

Consider the quadratic equation:

$$ax^2 + bx + c = 0$$

Its solutions are given by:

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

However, naïvely implementing this formula can lead to numerical problems such as **overflow**, **underflow**, and **catastrophic cancellation**, especially when the coefficients are very large or very small in magnitude.

- When b^2 is much larger than $4ac$, the discriminant $\sqrt{b^2 - 4ac}$ is nearly equal to $|b|$. Subtracting two nearly equal numbers, can cause significant loss of precision due to catastrophic cancellation.
- To compute the roots more accurately, use the following rearranged formula:

  $$x = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}}$$

  which has the flipped sign in the denominator and avoids the subtraction of two nearly equal numbers.

Function evaluations: When implementing a function for evaluation, take care in how you write it to avoid numerical problems:

E.g. when evaluating $f(x) = \sqrt{x+1} - \sqrt{x}$ for large x both terms are nearly equal, and their subtraction leads to loss of significant digits.

This can be solved by re-writing it as

$$f(x) = \frac{(\sqrt{x+1} - \sqrt{x})(\sqrt{x+1} + \sqrt{x})}{\sqrt{x+1} + \sqrt{x}} = \frac{1}{\sqrt{x+1} + \sqrt{x}}$$

```
x = 1e16  # Large value of x

# Original expression
f_original = math.sqrt(x + 1) - math.sqrt(x)

# alternative expression
f_alt = 1 / (math.sqrt(x + 1) + math.sqrt(x))

print("Original function result:", f_original)
print("Alternative function result:", f_alt)
```

```
Original function result: 0.0   Want minder dan 10^{-17} verschil
Alternative function result: 5e-09
```

CHAPTER 2

Systems of Linear Equations

```
from timeit import default_timer as timer

import matplotlib.pyplot as plt
import numpy as np
import schemdraw
import schemdraw.elements as elm
from scipy import linalg, optimize
```

2.1 Introduction and Notation

Many physical systems show *linear* behaviour:

- Newton's second law of motion $F = ma$
- Ohm's law : $R = \frac{U}{I}$
- Hooke's law: $F_s = kx$
- …

In some cases, we have an entire *system of coupled equations* which relate several "causes" with corresponding "effects". For instance, in an electrical circuit containing many conductors, Ohm's and Kirchoff's laws allow us to write a system of equations describing the relationship between the voltages and currents.

In general, such a system of m linear equations with n can be written as:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_n \end{cases}$$

We transform this into a matrix notation by expressing e.g. all the currents in a vector $\mathbf{x}$ and all voltages in a vector $\mathbf{b}$. The linear transformation between these two vector spaces is represented by a *matrix* $\mathbf{A}$.

Thus, a system of m linear equations with n unknowns can be transformed into a $m \times n$ matrix:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Such a relation can be used to predict $\mathbf{b}$ by a matrix-vector multiplication $\mathbf{b} = \mathbf{A}\mathbf{x}$.

More interestingly, we can ask the question: "if we know $\mathbf{b}$, can we then reconstruct $\mathbf{x}$?

In this chapter, we will answer this question. Here, we will assume that we have an equal amount of *unknowns* as we have *equations*, i.e. a *square* system.

Example: Electrical Circuit

Consider the electrical circuit shown below, consisting of two voltage sources and three resistances. We wish to determine the currents in the loops in the circuit by applying

Ohm's law: The voltage drop across a resistance in the direction of a current I is given by IR.

Kirchhoff's voltage law: The net voltage drop in a closed loop is zero.

and

Kirchhoff's current law: For every junction in a circuit the sum of currents flowing in is equal to the currents flowing out

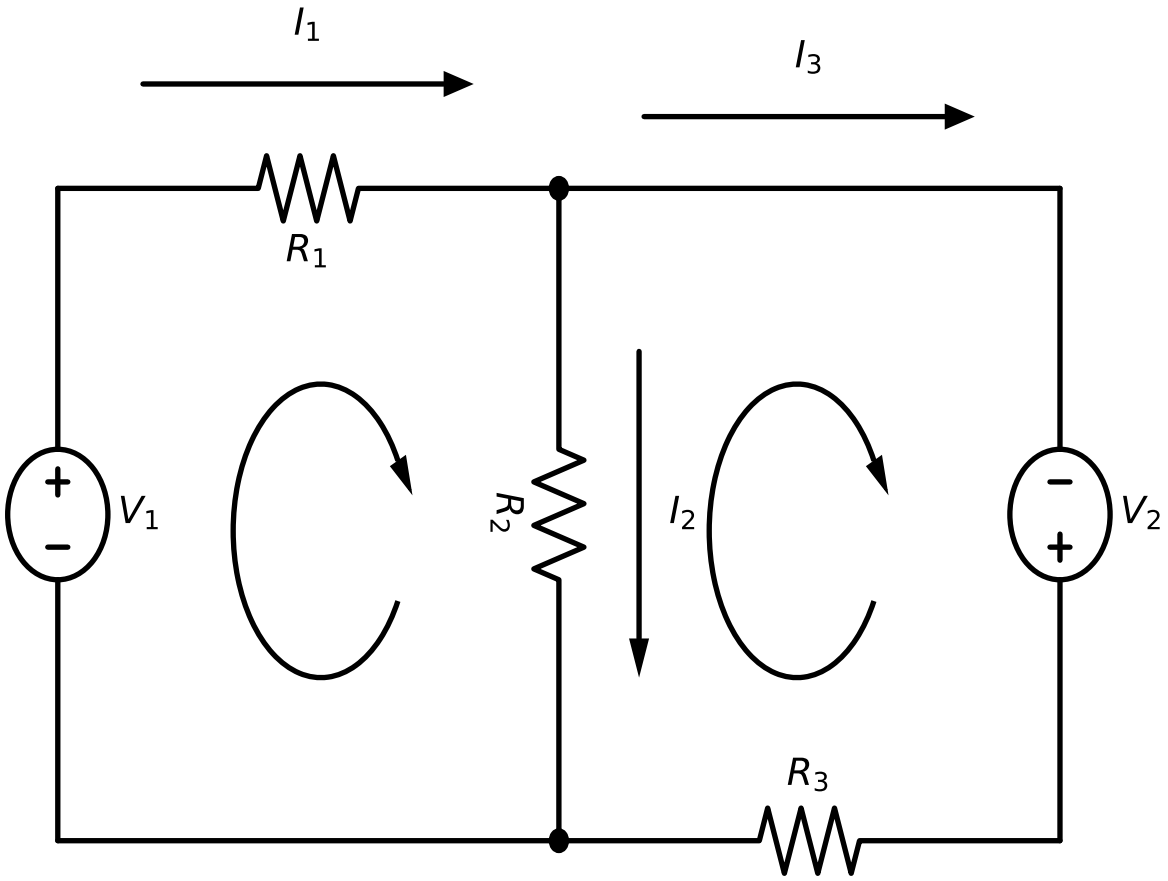
> Note on the figure below. The arrow indicating the current direction labeled I_2 should point down. If this is not the case for you, you might have an older version of schemdraw. You can update with `pip install -U schemdraw`. Version 0.17 (or newer) gives the correct result.

```
def draw_circuit():
    plt.close("circuit")
    fig, ax = plt.subplots(num="circuit")
    ax.set_axis_off()
    with schemdraw.Drawing(ax, unit=5) as d:
        V1 = elm.SourceV().label("$V_1$", loc="bot")
        d += V1
        R1 = elm.Resistor().right().label("$R_1$", loc="bot")
        d += R1
        d += elm.CurrentLabel(length=3.3, ofst=0.55).at(R1).label("$I_1$")
        d += elm.Dot()
        d.push()
        R2 = elm.Resistor().down().label("$R_2$", rotate=True)
        d += R2
        d += elm.CurrentLabel(top=True, length=2.5, ofst=0.55).at(R2).label("
↪$I_2$")
        d += elm.Dot()
        d.pop()
        L1 = elm.Line()
        d += L1
        V2 = elm.SourceV().down().label("$V_2$", loc="bot")
        d += V2
        R3 = elm.Resistor().left().label("$R_3$")
        d += R3
        L2 = elm.Line().left().tox(V1.start)
        d += L2
```

(continues on next page)

(continued from previous page)

```
        d += elm.CurrentLabel(length=3.3, ofst=0.55).at(L1).label("$I_3$")
        d += elm.LoopCurrent([R1, R2, L2, V1], pad=1.25)
        d += elm.LoopCurrent([R1, V2, L2, R2], pad=1.25)


draw_circuit()
```



Applying these laws to each loop in the current results in the following system of three equations:

$$\begin{cases} I_1 - I_2 - I_3 = 0 \\ I_1 R_1 + I_2 R_2 = V_1 \\ -I_2 R_2 + I_3 R_3 = V_2 \end{cases}$$

which can be written in matrix form as

$$\begin{bmatrix} 1 & -1 & -1 \\ R_1 & R_2 & 0 \\ 0 & -R_2 & +R_3 \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \\ I_3 \end{bmatrix} = \begin{bmatrix} 0 \\ V_1 \\ V_2 \end{bmatrix}$$

With the methods learned below, you'll be able to solve this system.

An $n \times n$ matrix is called *nonsingular* if it satisfies any one of the following equivalent conditions:

- $\mathbf{A}$ has an inverse $\mathbf{A}^{-1}$ such that $\mathbf{A}^{-1}\mathbf{A}=\mathbf{I}$ (the identity matrix)
- $\det(\mathbf{A}) \neq 0$
- $\text{rank}(\mathbf{A}) = n$ (the **rank** of matrix is the maximum number of linearly independent rows or columns it contains)

- for any vector $\mathbf{z} \neq 0$, $\mathbf{Az}$ also must be nonzero.

A system with a nonsingular matrix always has one unique solution $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. A system with a singular matrix either has no solution or infinitely many solutions, depending on the vector $\mathbf{b}$.

In two dimensions this is easy to visualize. Each linear equation represents a straight line in the plane. The solution of the system is the intersection between those lines. *If* they are not parallel, they have a unique intersection point (the nonsingular case). If they are parallel they either do not intersect at all (no solution) or they coincide (infinitely many solutions).

2.2 Solving Linear Systems

2.2.1 Strategy

To solve a linear system $\mathbf{Ax} = \mathbf{b}$, our general strategy is to transform our problem in another one whose solution is easier to compute. To this end we can premultiply (multiply from the left) both sides by *any* nonsingular matrix M, without affecting our solution:

To see why, note that the solution to the system

$$\mathbf{MAz} = \mathbf{Mb}$$

is given by

$$\mathbf{z} = (\mathbf{MA})^{-1}\mathbf{Mb} = \mathbf{A}^{-1}\mathbf{M}^{-1}\mathbf{Mb} = \mathbf{A}^{-1}\mathbf{b} = \mathbf{x}$$

> **Example**
>
> A simple example of such a transformation is a permutation $\mathbf{P}$ which reorders the rows of $\mathbf{A}$ and the corresponding entries of $\mathbf{b}$. It's easy to see that this does not change the solution $\mathbf{x}$ as all of the equations in the system must be satisfied simultaneously, so the order in which they are written down is irrelevant.
>
> The permutation matrix $\mathbf{P}$ is a square matrix with exactly one 1 in each row and zeros elsewhere.
>
> e.g.
>
> $$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} x_3 \\ x_1 \\ x_2 \end{bmatrix}$$

The next question we need to answer is what systems are computationally easy to solve?

The answer to this question is a **triangular linear system**.

Such a system contains one row with only one nonzero entry, such that one of the unknowns can easily be determined. The system also contains a row with two nonzero entries, one of which is already determined, and so on… A matrix with these properties is called a **triangular matrix** and is easy to solve by successive substitution. Note that only square or $n \times n$ matrices can be triangular.

Although the general triangular form described above is all that is required to enable the system to be solved by successive substitution, it is convenient to define two specific triangular forms for computational purposes: lower and upper triangular matrices. Any triangular matrix can be permuted into upper or lower triangular form by a suitable permutation of its rows or columns.

A matrix $\mathbf{L}$ is **lower triangular** if all of its entries above the main diagonal are zero ($l_{ij} = 0$ if $i < j$):

$$\begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

A lower triangular system $\mathbf{Lx} = \mathbf{b}$ can be solved by **forward substitution**, mathematically expressed as

$$x_1 = b_1/l_{11}, \quad x_i = \left(b_i - \sum_{j=1}^{i-1} l_{ij}x_j\right)/l_{ii} \quad \text{for} \quad i = 2, \ldots, n$$

> **Example**
>
> As an example let's use this approach on the lower triangular system:
>
> $$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 0.5 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 2 \end{bmatrix}$$
>
> For such a simple system, we can easily calculate by hand that the solution is:
>
> $x_1 = 2$
>
> $x_2 = \frac{4-(2\cdot 2)}{1} = 0$
>
> $x_3 = \frac{2-(-1\cdot 2+0\cdot 0.5)}{-1} = -4$
>
> The following cell shows an implementation which does this automatically to confirm our answer:

```
def forward_substitution(L, b):
    n = len(L)
    x = np.zeros(n)
    for j in range(n):
        if L[j][j] == 0:  # stop if matrix is singular
            break
        x[j] = b[j] / L[j][j]
        for i in range(j, n):
            b[i] = b[i] - L[i][j] * x[j]
    return x


L = np.array([[1, 0, 0], [2, 1, 0], [-1, 0.5, -1]])
b = np.array([2, 4, 2])

x = forward_substitution(L, b)
x
```

```
array([ 2.,  0., -4.])
```

A matrix $\mathbf{U}$ is **upper triangular** if all of its entries *below* the main diagonal are zero ($u_{ij} = 0$ if $i > j$):

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}$$

An upper triangular system $\mathbf{Ux} = \mathbf{b}$ can be solved by **backward substitution**, mathematically expressed as

$$x_n = b_n/u_{nn}, \quad x_i = \left(b_i - \sum_{j=1+i}^{n} u_{ij}x_j\right)/u_{ii} \quad \text{for} \quad i = n-1, \ldots, 1$$

This can similarly be solved as illustrated on the following example:

Example

As an example let us use this approach on the lower triangular system:

$$\begin{bmatrix} 1 & 2 & 2 \\ 0 & -4 & -6 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ -6 \\ 1 \end{bmatrix}$$

$x_3 = -1$

$x_2 = \frac{-6-(-6\cdot(-1))}{-4} = 3$

$x_1 = \frac{3-(2\cdot 3+2\cdot(-1))}{1} = -1$

The following cell shows an implementation which does this automatically to confirm our answer:

```
def backward_substitution(U, b):
    n = len(U)
    x = np.zeros(n)
    # Note that he last value of range is exclusive,
    # which is very counter-intuitive for countdowns.
    for j in range(n - 1, -1, -1):
        if U[j][j] == 0:  # stop if matrix is singular
            break
        x[j] = b[j] / U[j][j]
        for i in range(0, j):
            b[i] = b[i] - U[i][j] * x[j]
    return x


U = np.array([[1, 2, 2], [0, -4, -6], [0, 0, -1]])
b = np.array([3, -6, 1])

x = backward_substitution(U, b)
x
```

```
array([-1.,  3., -1.])
```

2.2.2 Elementary elimination matrices

What we need is a nonsingular linear transformation that transforms a given linear system into a triangular linear system, which we can subsequently solve.

Thus, we need a transformation that replaces selected nonzero entries of the given matrix with zeros. This can be accomplished by taking appropriate linear combinations of the rows of the matrix

Generally, for an n-vector $\mathbf{a}$, we can annihilate *all* of its entries below the kth position (assuming $a_k \neq 0$) by the following transformation:

$$\mathbf{M}_k\mathbf{a} = \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & \cdots & -m_{k+1} & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & -m_n & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_k \\ a_{k+1} \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} a_1 \\ \vdots \\ a_k \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

with $m_i = \frac{a_i}{a_k} \quad i = k+1, \cdots, n$

The divisor a_k is called the **pivot**. A matrix of this form is called an **elementary elimination matrix** or **Gauss transformation** and its effect on a vector is to add a multiple of row k to each subsequent row with the multipliers m_i chosen so that the result in each case is zero.

Useful properties:

- $\mathbf{M}_k = \mathbf{I} - \mathbf{m}_k \mathbf{e}_k^\intercal$, where $\mathbf{m}_k = [0, \cdots, 0, m_{k+1}, \cdots, m_n]^\intercal$ and $\mathbf{e}_k$ is the kth column of the identity matrix
- $\mathbf{M}_k^{-1} = \mathbf{I} + \mathbf{m}_k \mathbf{e}_k^\intercal$, which means that $\mathbf{M}_k^{-1}$, denoted as $\mathbf{L}_k$, is the same as $\mathbf{M}_k$, except that the signs of the multipliers are reversed.

> **Example**
>
> if $\mathbf{a} = [2, 4, -2]^\intercal$
>
> $$\mathbf{M}_1\mathbf{a} = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \\ -2 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix}$$
>
> and
>
> $$\mathbf{M}_2\mathbf{a} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0.5 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \\ -2 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 0 \end{bmatrix}$$

2.2.3 Gaussian Elimination and LU Factorization

Using elementary elimination matrices, it is relatively easy to reduce a general linear system $\mathbf{Ax} = \mathbf{b}$ to upper triangular form, using the following procedure called **Gaussian elimination**, **LU factorization** or **LU decomposition**:

- We first choose an elementary elimination matrix $\mathbf{M}_1$, with the first diagonal entry of $\mathbf{A} = a_{11}$ as the pivot, so that the first column of $\mathbf{M}_1\mathbf{A}$ becomes zero under the top element. The new system becomes $\mathbf{M}_1\mathbf{Ax} = \mathbf{M}_1\mathbf{b}$, which has the same solution.
- Next we use the second diagonal entry of $\mathbf{M}_1\mathbf{Ax}$ as pivot to determine $\mathbf{M}_2$, so that the second column of $\mathbf{M}_2\mathbf{M}_1\mathbf{A}$ becomes zero under the second element. The zeros in the first column under the top element are preserved. We obtain the system $\mathbf{M}_2\mathbf{M}_1\mathbf{Ax} = \mathbf{M}_2\mathbf{M}_1\mathbf{b}$.
- We keep doing this until the matrix is upper triangular.
- If we define the matrix $\mathbf{M} = \mathbf{M}_{n-1} \cdots \mathbf{M}_1$, then the transformed linear system $\mathbf{MAx} = \mathbf{Mb}$ is upper triangular and can be solved by back-substitution to obtain the solution of the original linear system $\mathbf{Ax} = \mathbf{b}$.

The name LU factorization comes from the fact that it decomposes the matrix $\mathbf{A}$ into a product of a lower triangular matrix $\mathbf{L} = \mathbf{M}^{-1}$, and an upper triangular matrix $\mathbf{U} = \mathbf{MA}$.

Given such a factorization, the linear system $\mathbf{Ax} = \mathbf{b}$ can be written as $\mathbf{LUx} = \mathbf{b}$ and can be solved by first solving the lower triangular system $\mathbf{Ly} = \mathbf{b}$ by forward substitution, then the upper triangular system $\mathbf{Ux} = \mathbf{y}$ by back-substitution.

Note that the factorization phase can be skipped when solving additional systems having different right-hand-side vectors $\mathbf{b}$ but the same matrix $\mathbf{A}$, since $\mathbf{L}$ and $\mathbf{U}$ can be reused.

> **Example**
>
> As an example, let's solve the linear system
>
> $$\begin{aligned} 2x_1 + 7x_2 + 5x_3 &= 6 \\ -2x_1 - 3x_2 - x_3 &= 18 \\ 4x_1 + 9x_2 - 5x_3 &= 12 \end{aligned}$$

In matrix notation this becomes

$$\mathbf{Ax} = \begin{bmatrix} 2 & 7 & 5 \\ -2 & -3 & -1 \\ 4 & 9 & -5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6 \\ 18 \\ 12 \end{bmatrix} = \mathbf{b}$$

Using

$$\mathbf{M}_1 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ -2 & 0 & 1 \end{bmatrix},$$

we find

$$\mathbf{M}_1\mathbf{A} = \begin{bmatrix} 2 & 7 & 5 \\ 0 & 4 & 4 \\ 0 & -5 & -15 \end{bmatrix}$$

and

$$\mathbf{M}_1\mathbf{b} = \begin{bmatrix} 6 \\ 24 \\ 0 \end{bmatrix}$$

Using

$$\mathbf{M}_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & \frac{5}{4} & 1 \end{bmatrix},$$

we find

$$\mathbf{M}_2\mathbf{M}_1\mathbf{A} = \begin{bmatrix} 2 & 7 & 5 \\ 0 & 4 & 4 \\ 0 & 0 & -10 \end{bmatrix}$$

and

$$\mathbf{M}_2\mathbf{M}_1\mathbf{b} = \begin{bmatrix} 6 \\ 24 \\ 30 \end{bmatrix}.$$

By using $\mathbf{M} = \mathbf{M}_2\mathbf{M}_1$ and $\mathbf{U} = \mathbf{MA}$, we have therefore reduced the original system to the equivalent upper triangular system

$$\mathbf{Ux} = \begin{bmatrix} 2 & 7 & 5 \\ 0 & 4 & 4 \\ 0 & 0 & -10 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6 \\ 24 \\ 30 \end{bmatrix} = \mathbf{Mb} = \mathbf{y},$$

which can be solved by back-substitution, as shown in a previous example.

Thus, the matrices L_1 and L_2 can be obtained from M_1 and M_2 according to the properties:

$$\mathbf{L}_1 = \mathbf{M}_1^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 2 & 0 & 1 \end{bmatrix}$$

and

$$\mathbf{L}_2 = \mathbf{M}_2^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -\frac{5}{4} & 1 \end{bmatrix}$$

We can also write down the LU-factorization explicitly with

$$\mathbf{L} = \mathbf{L}_1\mathbf{L}_2 = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 2 & -\frac{5}{4} & 1 \end{bmatrix}$$

and

$$\mathbf{A} = \mathbf{I}\mathbf{A} = (\mathbf{M}_1^{-1}\mathbf{M}_2^{-1}\mathbf{M}_2\mathbf{M}_1)\mathbf{A} = (\mathbf{M}_1^{-1}\mathbf{M}_2^{-1})(\mathbf{M}_2\mathbf{M}_1\mathbf{A}) = (\mathbf{L}_1\mathbf{L}_2)\mathbf{U} = \mathbf{L}\mathbf{U}.$$

As a result, we confirm that

$$\mathbf{A} = \begin{bmatrix} 2 & 7 & 5 \\ -2 & -3 & -1 \\ 4 & 9 & -5 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 2 & -\frac{5}{4} & 1 \end{bmatrix} \begin{bmatrix} 2 & 7 & 5 \\ 0 & 4 & 4 \\ 0 & 0 & -10 \end{bmatrix} = \mathbf{L}\mathbf{U}$$

An algorithm to perform LU factorization by Gaussian elimination is shown below. A note on the variable names used: it returns **L** and **U**, despite using variables **M** and **A** in the algorithm!

```
def LU(A, verbose=False):
    """LU decomposition with Gaussian elimination.

    Parameters
    ----------
    A: np.array
        2d-square array.
    verbose: bool, default=False
        Whether print more infomation.

    Returns
    -------
    tuple
        (L,U)
    """
    U = np.copy(A)
    n = U.shape[0]
    L = np.identity(n)

    # loop columns
    for j in range(n):
        M_jth, L_jth = np.identity(n), np.identity(n)
        if U[j, j] == 0:  # stop if pivot is zero
            continue

        # loop rows
        for i in range(j + 1, n):
            M_jth[i, j] = -U[i, j] / U[j, j]
            L_jth[i, j] = -M_jth[i, j]

        if verbose:
```

(continues on next page)

(continued from previous page)

```
            print(f" No.{j+1} iteration ".center(80, "*"))
            print("M: ")
            print(M_jth)
            print("L: ")
            print(L_jth)
            print("Current A")
            print(U)
            print(" end ".center(80, "-"))

        # update A, L, U
        L = L @ L_jth
        U = M_jth @ U
    return L, U
```

```
print("As an example consider the matrix A from the previous example:")
A = np.array([[2, 7, 5], [-2, -3, -1], [4, 9, -5]])
print(A)
print()

L, U = LU(A, verbose=True)
assert np.allclose(L @ U, A)

print("The matrices L and U read")
print(L)
print("and")
print(U)
print()
print("Let's check that their product indeed equals A")
print(L @ U)
print("For a given vector b, this system can now be solved as follows:")
b = np.array([6, 18, 12])

y = forward_substitution(L, b)
x = backward_substitution(U, y)
x
```

```
As an example consider the matrix A from the previous example:
[[ 2  7  5]
 [-2 -3 -1]
 [ 4  9 -5]]

******************************** No.1 iteration␣
 ↪********************************
M:
[[ 1.  0.  0.]
 [ 1.  1.  0.]
 [-2.  0.  1.]]
L:
[[ 1.  0.  0.]
 [-1.  1.  0.]
 [ 2.  0.  1.]]
Current A
[[ 2  7  5]
```

(continues on next page)

(continued from previous page)

```
 [-2 -3 -1]
 [ 4  9 -5]]
------------------------------------ end ------------------------------------
 ↪--
******************************** No.2 iteration␣
 ↪********************************
M:
[[1.   0.   0.  ]
 [0.   1.   0.  ]
 [0.   1.25 1.  ]]
L:
[[ 1.    0.    0.  ]
 [ 0.    1.    0.  ]
 [ 0.   -1.25  1.  ]]
Current A
[[  2.   7.   5.]
 [  0.   4.   4.]
 [  0.  -5. -15.]]
------------------------------------ end ------------------------------------
 ↪--
******************************** No.3 iteration␣
 ↪********************************
M:
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
L:
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
Current A
[[  2.   7.   5.]
 [  0.   4.   4.]
 [  0.   0. -10.]]
------------------------------------ end ------------------------------------
 ↪--
The matrices L and U read
[[ 1.    0.    0.  ]
 [-1.    1.    0.  ]
 [ 2.   -1.25  1.  ]]
and
[[  2.   7.   5.]
 [  0.   4.   4.]
 [  0.   0. -10.]]

Let's check that their product indeed equals A
[[ 2.  7.  5.]
 [-2. -3. -1.]
 [ 4.  9. -5.]]
For a given vector b, this system can now be solved as follows:
```

```
array([-21.,   9.,  -3.])
```

2.2.4 Partial pivoting

There are 2 problems with the Gaussian elimination process:

1. The process breaks down if the leading diagonal entry of the matrix is zero at any stage because the multipliers for a given column require a division by the diagonal entry in that column.

- This issue is trivial to solve: when the diagonal entry is zero at stage k, then interchange row k of the system (and the right-hand side vector) with some subsequent row with nonzero entry in column k. Such an interchange does not change the solution of the system and is called **pivoting**.

2. In finite-precision arithmetic we wish to limit the size of the multipliers so that previous rounding errors do not get amplified.

- The multipliers will never exceed 1 in magnitude if for each column we choose the entry of the largest magnitude on or below the diagonal as a pivot. Such a policy is called **partial pivoting** and is essential in practice for a numerically stable implementation of Gaussian elimination.

> **Example**
>
> In the following example we'll perform the LU factorization for a matrix $\mathbf{A}$ without pivoting
>
> $$\mathbf{A} = \begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix}$$
>
> and with pivoting, i.e. with matrix
>
> $$\mathbf{A}' = \begin{bmatrix} 1 & 1 \\ \epsilon & 1 \end{bmatrix}$$

> We'll see that for $\epsilon = 1 \times 10^{-16}$ (about the machine precision for doubles) the product $\mathbf{LU}$ no longer equals $\mathbf{A}$, whereas it gives the correct answer for $\mathbf{A}'$.

```
e = 1e-16
a = np.array([[e, 1], [1, 1]])
aprime = np.array([[1, 1], [e, 1]])

L, U = LU(a)
Lprime, Uprime = LU(aprime)

print(L @ U)
print(Lprime @ Uprime)
```

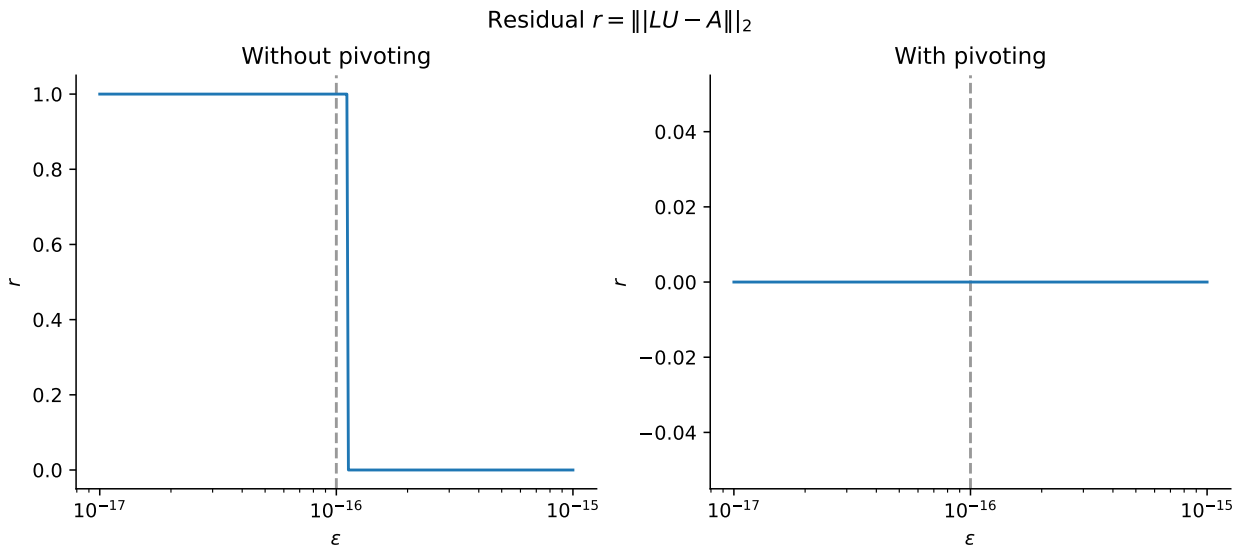
```
[[1.e-16 1.e+00]
 [1.e+00 0.e+00]]
[[1.e+00 1.e+00]
 [1.e-16 1.e+00]]
```

```
def compute_residual(eps_arr):
    res = np.zeros((len(eps_arr), 2))
    for i, eps in enumerate(eps_arr):
        a = np.array([[eps, 1], [1, 1]])
        aprime = np.array([[1, 1], [eps, 1]])
        L, U = LU(a)
        Lprime, Uprime = LU(aprime)
```

(continues on next page)

(continued from previous page)

```
        res[i, 0] = linalg.norm(L @ U - a, 2)
        res[i, 1] = linalg.norm(Lprime @ Uprime - aprime, 2)
    return res


def plot_residual():
    plt.close("residual")
    fig, axs = plt.subplots(ncols=2, figsize=(9, 4), num="residual")
    for ax in axs:
        ax.set_xscale("log")
        ax.set_xlabel(r"$\epsilon$")
        ax.set_ylabel("$r$")
        ax.axvline(10 ** (-16), 0, 1, color="k", alpha=0.4, linestyle="--")

    eps_arr = np.linspace(10 ** (-17), 10 ** (-15), 600)
    res = compute_residual(eps_arr)

    axs[0].plot(eps_arr, res[:, 0])
    axs[0].set_title("Without pivoting")
    axs[1].plot(eps_arr, res[:, 1])
    axs[1].set_title("With pivoting")
    fig.suptitle(r"Residual $r=\|| LU - A \||_2$")


plot_residual()
```



The previous example is rather artificial, but in general, larger pivots result in smaller entries in the elimination matrix and hence smaller errors.

The Gaussian elimination procedure becomes a little more complicated as each elimination matrix is now preceded by a permutation matrix that interchanges rows to maximize the pivot value.

> **Example**
>
> Let's now repeat the previous example, but with partial pivoting.

We again start from

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} 2 & 7 & 5 \\ -2 & -3 & -1 \\ 4 & 9 & -5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6 \\ 18 \\ 12 \end{bmatrix} = \mathbf{b}$$

The largest entry in the first column is 4, so we interchange the first and last row using

$$\mathbf{P}_1 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

and obtain

$$\mathbf{P}_1\mathbf{A}\mathbf{x} = \begin{bmatrix} 4 & 9 & -5 \\ -2 & -3 & -1 \\ 2 & 7 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 12 \\ 18 \\ 6 \end{bmatrix} = \mathbf{P}_1\mathbf{b}$$

To annihilate the subdiagonal entries of the first column, we use

$$\mathbf{M}_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ -0.5 & 0 & 1 \end{bmatrix}$$

To obtain the transformed system

$$\mathbf{M}_1\mathbf{P}_1\mathbf{A}\mathbf{x} = \begin{bmatrix} 4 & 9 & -5 \\ 0 & 1.5 & -3.5 \\ 0 & 2.5 & 7.5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 12 \\ 24 \\ 0 \end{bmatrix} = \mathbf{M}_1\mathbf{P}_1\mathbf{b}$$

Similarly, we exchange the last two rows using $\mathbf{P}_2$ and use the elimination matrix $\mathbf{M}_2$:

$$\mathbf{P}_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad \text{and,} \quad \mathbf{M}_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -0.6 & 1 \end{bmatrix}$$

To obtain the transformed system

$$\mathbf{U}\mathbf{x} = \mathbf{M}_2\mathbf{P}_2\mathbf{M}_1\mathbf{P}_1\mathbf{A}\mathbf{x} = \begin{bmatrix} 4 & 9 & -5 \\ 0 & 2.5 & 7.5 \\ 0 & 0 & -8 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 12 \\ 0 \\ 24 \end{bmatrix} = \mathbf{M}_2\mathbf{P}_2\mathbf{M}_1\mathbf{P}_1\mathbf{b}$$

which can be solved by back-substitution.

Again, we can write out the LU factorization explicitly, using

$$\mathbf{L} = \mathbf{M}^{-1} = (\mathbf{M}_2\mathbf{P}_2\mathbf{M}_1\mathbf{P}_1)^{-1} = \mathbf{P}_1^\mathsf{T}\mathbf{L}_1\mathbf{P}_2^\mathsf{T}\mathbf{L}_2 =$$

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ -0.5 & 1 & 0 \\ 0.5 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0.6 & 1 \end{bmatrix} = \begin{bmatrix} 0.5 & 1 & 0 \\ -0.5 & 0.6 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

such that

$$\mathbf{A} = \begin{bmatrix} 2 & 7 & 5 \\ -2 & -3 & -1 \\ 4 & 9 & -5 \end{bmatrix} = \begin{bmatrix} 0.5 & 1 & 0 \\ -0.5 & 0.6 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 4 & 9 & -5 \\ 0 & 2.5 & 7.5 \\ 0 & 0 & -8 \end{bmatrix} = \mathbf{L}\mathbf{U}$$

Note that **L** is not longer lower diagonal, but still is a triangular matrix in the general sense that it is a permutation of a lower triangular matrix. There also exist alternative ways to write **L** depending on whether **P** is used to permute **A** or **L** itself. The notation used here is consistent with what is used in `scipy.linalg` with the option `permute_l=True`.

The following cell shows this algorithm which performs LU-factorization, including partial pivoting, resulting in **LU** equals matrix **A**:

```
def LU_pivot(A, verbose=False):
    """LU decomposition with Gaussian elimination using permute.

    Parameters
    ----------
    A: np.array
        2d-square array.
    verbose: bool, default=True
        Whether print more infomation.

    Returns
    -------
    tuple
        (L,U)

    """
    U = np.copy(A)
    n = U.shape[0]
    L = np.identity(n)

    # loop columns
    for j in range(n):
        M_jth, L_jth, P_jth = (
            np.identity(n),
            np.identity(n),
            np.identity(n, dtype=int),
        )
        if U[j, j] == 0:  # stop if pivot is zero
            continue

        # look for the largest pivot
        index_largest = j
        for i in range(j + 1, n):
            if U[i][j] > U[index_largest][j]:
                index_largest = i

        if j != index_largest:  # swap rows
            U[[j, index_largest]] = U[[index_largest, j]]
            P_jth[[j, index_largest]] = P_jth[[index_largest, j]]

        # loop rows
        for i in range(j + 1, n):
            M_jth[i, j] = -U[i, j] / U[j, j]
            L_jth[i, j] = -M_jth[i, j]

        if verbose:
            print(f" No.{j+1} iteration ".center(80, "*"))
            print("P: ")
            print(P_jth)
```

(continues on next page)

(continued from previous page)

```
            print("M: ")
            print(M_jth)
            print("L: ")
            print(L_jth)
            print("Current A")
            print(U)
            print(" end ".center(80, "-"))

        # update A, L, U
        L = L @ P_jth.T @ L_jth
        U = M_jth @ U
    return L, U
```

```
A = np.array([[2.0, 7.0, 5.0], [-2.0, -3.0, -1.0], [4.0, 9.0, -5.0]])
L, U = LU_pivot(A, verbose=True)

print(L, "\n\n", U)
assert np.allclose(L @ U, A)
```

```
******************************** No.1 iteration␣
↪*******************************
P:
[[0 0 1]
 [0 1 0]
 [1 0 0]]
M:
[[ 1.   0.   0. ]
 [ 0.5  1.   0. ]
 [-0.5  0.   1. ]]
L:
[[ 1.   0.   0. ]
 [-0.5  1.   0. ]
 [ 0.5  0.   1. ]]
Current A
[[ 4.  9. -5.]
 [-2. -3. -1.]
 [ 2.  7.  5.]]
------------------------------------ end -------------------------------------
↪--
******************************** No.2 iteration␣
↪*******************************
P:
[[1 0 0]
 [0 0 1]
 [0 1 0]]
M:
[[ 1.   0.   0. ]
 [ 0.   1.   0. ]
 [ 0.  -0.6  1. ]]
L:
[[1.  0.  0. ]
 [0.  1.  0. ]
 [0.  0.6 1. ]]
```

(continues on next page)

(continued from previous page)

```
Current A
[[ 4.   9.  -5. ]
 [ 0.   2.5  7.5]
 [ 0.   1.5 -3.5]]
------------------------------------ end ----------------------------------
 ↪--
****************************** No.3 iteration␣
 ↪******************************
P:
[[1 0 0]
 [0 1 0]
 [0 0 1]]
M:
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
L:
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
Current A
[[ 4.   9.  -5. ]
 [ 0.   2.5  7.5]
 [ 0.   0.  -8. ]]
------------------------------------ end ----------------------------------
 ↪--
[[ 0.5  1.   0. ]
 [-0.5  0.6  1. ]
 [ 1.   0.   0. ]]

 [[ 4.   9.  -5. ]
 [ 0.   2.5  7.5]
 [ 0.   0.  -8. ]]
```

The name *partial* pivoting comes from the fact that only the current column is searched for a suitable pivot. **Complete pivoting** is a strategy in which the complete unreduced submatrix is searched for the largest entry, which is then permuted in the diagonal pivot position, which requires the interchanging of rows and columns. Without going into detail, it is worth noting that both in theory and in practice, the additional stability offered by complete pivoting is marginal and not worth the extra expense.

2.2.5 Gauss-Jordan elimination

The goal in Gaussian elimination is to reduce a matrix to triangular form, because the resulting system is easy to solve. A diagonal form is even easier to solve, so it might appear attractive to try to reduce a matrix to this form. **Gauss-Jordan elimination** is a variation of Gaussian elimination which does this by not only annihilating the matrix entries below the diagonal but also above it, making it about 50% more computationally expensive than standard Gaussian elimination.

Typically, it is not used because

- The final solution phase is computationally somewhat cheaper because of the diagonal form of the matrix, but this does not suffice make up for the additional cost in the elimination phase.

However, it might be desirable in some situations:

- In an implementation on parallel computers, the workload remains the same throughout the elimination phase and the final solutions can all be calculated at once.

- It can also be used to calculate the inverse of a matrix explicitly by initializing the right-hand side of the matrix as the identity matrix **I**.

Example: Mechanical System Equilibrium

Let us consider a system of three masses connected by springs in a vertical arrangement, subject to external forces. The equilibrium positions and forces can be expressed as follows:

$$\begin{cases} 3kx_1 - 2kx_2 = F_1 \\ -2kx_1 + (3k-k)x_2 - 3kx_3 = F_2 \\ -3kx_2 + 13kx_3 = F_3 \end{cases}$$

Here, x_1, x_2, and x_3 represent the displacements of the masses, and F_1, F_2, and F_3 are the external forces applied to each mass. Let's consider the case where $F_1 = 1k$, $F_2 = 2k$ and $F_3 = 3k$. After dividing through k, one can write this system of linear equations in matrix form:

$$\left[\begin{array}{ccc|c} 3 & -2 & 0 & 1 \\ -2 & 2 & -3 & 2 \\ 0 & -3 & 13 & 3 \end{array}\right]$$

After using the first diagonal element as pivot, `row1 /= 3`, one gets:

$$\left[\begin{array}{ccc|c} 1 & -\frac{2}{3} & 0 & \frac{1}{3} \\ -2 & 9 & -3 & 2 \\ 0 & -3 & 13 & 3 \end{array}\right]$$

The first elimination is similar to LU: `row2 += 2 * row1`

$$\left[\begin{array}{ccc|c} 1 & -\frac{2}{3} & 0 & \frac{1}{3} \\ 0 & \frac{2}{3} & -3k & \frac{8}{3} \\ 0 & -3 & 13 & 3 \end{array}\right]$$

Then one uses the second diagonal as pivot: `row2 /= (2/3)`

$$\left[\begin{array}{ccc|c} 1 & -\frac{2}{3} & 0 & \frac{1}{3} \\ 0 & 1 & -\frac{9}{2} & 4 \\ 0 & -3 & 13 & 3 \end{array}\right]$$

Followed by elimination of coefficientbs *above and below* the diagonal: `row3 += 3 * row2` and `row1 += (2/3)*row2`

$$\left[\begin{array}{ccc|c} 1 & 0 & -3 & 3 \\ 0 & 1 & -\frac{9}{2} & 4 \\ 0 & 0 & -\frac{1}{2} & 15 \end{array}\right]$$

Finally, one use the last element as pivot and eleminates the coefficients above with similar upates:

$$\left[\begin{array}{ccc|c} 1 & 0 & 0 & -87 \\ 0 & 1 & 0 & 131 \\ 0 & 0 & 1 & -30 \end{array}\right]$$

```
def gauss_jordan_elimination(A, b):
    augmented_matrix = np.hstack((A.astype(float), b.astype(float)))
    rows, cols = augmented_matrix.shape
```

(continues on next page)

(continued from previous page)

```
    result_matrices = [augmented_matrix.copy()]

    for pivot_row in range(rows):
        augmented_matrix[pivot_row] /= augmented_matrix[pivot_row, pivot_row]

        for other_row in range(rows):
            if other_row != pivot_row:
                factor = augmented_matrix[other_row, pivot_row]
                augmented_matrix[other_row] -= factor * augmented_
↪matrix[pivot_row]

        result_matrices.append(augmented_matrix.copy())

    return result_matrices


def demo_gauss_jordan():
    # Example usage
    A = np.array([[3, -2, 0], [-2, 2, -3], [0, -3, 13]])
    b = np.array([[1], [2], [3]])

    result_matrices = gauss_jordan_elimination(A, b)

    # Print each intermediate matrix
    for i, result_matrix in enumerate(result_matrices):
        print(f"After {i} iteration{'s' if i > 1 else ''}:")
        print(result_matrix)
        print()

    # Print the final solution for x
    print("Final solution for x:")
    print(result_matrices[-1][:, -1])


demo_gauss_jordan()
```

```
After 0 iteration:
[[ 3. -2.  0.  1.]
 [-2.  2. -3.  2.]
 [ 0. -3. 13.  3.]]

After 1 iteration:
[[ 1.         -0.66666667  0.          0.33333333]
 [ 0.          0.66666667 -3.          2.66666667]
 [ 0.         -3.         13.          3.        ]]

After 2 iterations:
[[ 1.   0.  -3.   3. ]
 [ 0.   1.  -4.5  4. ]
 [ 0.   0.  -0.5 15. ]]

After 3 iterations:
[[   1.    0.    0.  -87.]
 [   0.    1.    0. -131.]
```

(continues on next page)

(continued from previous page)

```
 [  -0.   -0.    1.  -30.]]

Final solution for x:
[ -87. -131.  -30.]
```

2.2.6 Modified problems

In many practical situations linear systems do not occur in isolation but as part of a sequence of related problems. For example, one may need to solve a sequence of linear systems $\mathbf{Ax} = \mathbf{b}$ having the same matrix $\mathbf{A}$ but different right-hand sides $\mathbf{b}$. After having solved the initial system once, the factor $\mathbf{LU}$ can be used to solve the additional systems with a cost of $\mathcal{O}(n^2)$ instead of $\mathcal{O}(n^3)$, which is a substantial saving of work.

Additionally, there exist strategies to solve problems in which the matrix $\mathbf{A}$ is modified, but these lie beyond the scope of this course.

2.3 Special types of linear systems

In the algorithms discussed so far, we have assumed that the matrix is **dense** meaning that (almost) all the entries are nonzero.

If this is not the case and the matrix has some special properties, often there exist more efficient (in terms of storage or calculation time) ways to solve the linear system. A few examples are:

- **Symmetric**: $\mathbf{A} = \mathbf{A}^\intercal$, i.e. $a_{ij} = a_{ji}$ for all i, j
- **Positive definite**: $\mathbf{x}^\intercal\mathbf{Ax} > 0$ for all $\mathbf{x} \neq 0$
- **Banded**: $a_{ij} = 0$ for all $|i - j| > \beta$, with β the **bandwidth** of $\mathbf{A}$
- **Sparse**: most entries of $\mathbf{A}$ are zero

We'll have a detailed look at how to solve symmetric positive definite systems, as we'll encounter such systems later in the course when looking at *Linear least squares methods*.

2.3.1 Symmetric positive definite systems: Cholesky factorization

If a matrix $\mathbf{A}$ is symmetric and positive definite, then an LU factorization can be arranged so that $\mathbf{U} = \mathbf{L}^\intercal$, meaning that $\mathbf{A} = \mathbf{LL}^\intercal$, where $\mathbf{L}$ is lower triangular.

This is known as **Cholesky factorization** of $\mathbf{A}$. An algorithm for computing it can be derived by equating the corresponding entries of $\mathbf{A}$ and $\mathbf{LL}^\intercal$ and then generating the entries of $\mathbf{L}$ in the correct order.

E.g. in the 2D case we have:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{12} & a_{22} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 \\ l_{21} & l_{22} \end{bmatrix} \begin{bmatrix} l_{11} & l_{21} \\ 0 & l_{22} \end{bmatrix} = \begin{bmatrix} l_{11}^2 & l_{11}l_{21} \\ l_{11}l_{21} & l_{21}^2 + l_{22}^2 \end{bmatrix}$$

This implies

- $l_{11} = \sqrt{a_{11}}$
- $l_{21} = a_{12}/l_{11}$
- $l_{22} = \sqrt{a_{22} - l_{21}^2}$

In the 3D case we have:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{12} & a_{22} & a_{23} \\ a_{13} & a_{23} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11}^2 & l_{11}l_{21} & l_{11}l_{31} \\ l_{11}l_{21} & l_{21}^2 + l_{22}^2 & l_{21}l_{31} + l_{22}l_{32} \\ l_{11}l_{31} & l_{31}l_{21} + l_{32}l_{22} & l_{31}^2 + l_{32}^2 + l_{33}^2 \end{bmatrix}$$

This implies

- $l_{11} = \sqrt{a_{11}}$
- $l_{21} = a_{21}/l_{11}$
- $l_{31} = a_{31}/l_{11}$
- $l_{22} = \sqrt{a_{22} - l_{21}^2}$
- $l_{32} = (a_{23} - l_{31}l_{21})/l_{22}$
- $l_{33} = \sqrt{a_{33} - l_{31}^2 - l_{32}^2}$

The Cholesky factorization has a few very attractive properties:

- The n square roots are all of positive numbers, so the algorithm is well-defined
- Pivoting is not required
- Only the lower triangle of $\mathbf{A}$ is accessed, and hence the strict upper triangular portion need not be stored
- Only about $n^3/6$ multiplications and a similar number of additions are required.

Thus Cholesky factorization requires only about half as much storage and work as general LU-factorization.

A python algorithm to perform Cholesky factorization is shown in the cell below.

(This code is derived from https://www.quantstart.com/articles/Cholesky-Decomposition-in-Python-and-NumPy/)

```
def cholesky(A):
    """Perform a Cholesky decomposition of A.

    The given matrix must be a symmetric and positive definite.
    The function returns the lower variant triangular matrix, L.
    """

    n = len(A)

    # Create zero matrix for L
    L = [[0.0] * n for i in range(0, n)]

    # Perform the Cholesky decomposition
    for i in range(0, n):
        for k in range(0, i + 1):
            tmp_sum = sum(L[i][j] * L[k][j] for j in range(0, k))

            if i == k:
                L[i][k] = np.sqrt(A[i][i] - tmp_sum)
            else:
                L[i][k] = 1.0 / L[k][k] * (A[i][k] - tmp_sum)
    return np.array(L)
```

Example

As an example we'll compute the Cholesky factorization of the following symmetric positive definite matrix

$$\mathbf{A} = \begin{bmatrix} 4 & 2 & 6 \\ 2 & 2 & 5 \\ 6 & 5 & 22 \end{bmatrix}$$

We'll only show the lower triangle of the matrix, as this contains all information

First, we divide the first column by the square root of its diagonal entry

$$\begin{bmatrix} 2 & & \\ 1 & 2 & \\ 3 & 5 & 22 \end{bmatrix}$$

Next, we update the second column by subtracting the (2,1) entry from it, multiplied by the relevant entry in the first column

$$\begin{bmatrix} 2 & & \\ 1 & 2-(1\cdot 1) & \\ 3 & 5-(1\cdot 3) & 22 \end{bmatrix} = \begin{bmatrix} 2 & & \\ 1 & 1 & \\ 3 & 2 & 22 \end{bmatrix}$$

The last column is updated by subtracting the (3,1) entry from it, also times the relevant portion of the first column:

$$\begin{bmatrix} 2 & & \\ 1 & 1 & \\ 3 & 2 & 22-(3\cdot 3) \end{bmatrix} = \begin{bmatrix} 2 & & \\ 1 & 1 & \\ 3 & 2 & 13 \end{bmatrix}$$

The second column is then divided by the square root of its diagonal entry

$$\begin{bmatrix} 2 & & \\ 1 & 1 & \\ 3 & 2 & 13 \end{bmatrix}$$

and the last column is updated by subtracting the (3,2) entry multiplied by the relevant portion of the second column:

$$\begin{bmatrix} 2 & & \\ 1 & 1 & \\ 3 & 2 & 13-(2\cdot 2) \end{bmatrix}$$

Finally, the taking the square root of the third diagonal entry yields the final result

$$\mathbf{L} = \begin{bmatrix} 2 & & \\ 1 & 1 & \\ 3 & 2 & 3 \end{bmatrix}$$

```
# Let's now solve the same problem using the python algorithm shown above:
a = np.array([[4, 2, 6], [2, 2, 5], [6, 5, 22]])
L = cholesky(a)

print(L, "\n\n")

# check that L L^\intercal indeed equals a
print(L @ L.transpose())
```

```
[[2. 0. 0.]
 [1. 1. 0.]
 [3. 2. 3.]]


[[ 4.  2.  6.]
 [ 2.  2.  5.]
 [ 6.  5. 22.]]
```

2.3.2 Computational complexity

As shown in the examples below:

- LU factorization of an $n \times n$ matrix takes about $n^3/3$ floating point operations (flops)
- A complete matrix inversion takes about n^3 flops and thus is 3 times as expensive
- Solving an LU-factorized system using forward and backward substitution takes about n^2 flops. For large systems, this is negligible compared to the factorization phase.
- Cramer's rule (in which the system is solved using ratios of determinants) is astronomically expensive

In practice, inverting a matrix should almost never be done, because it is computationally too expensive and even results in less accuracy than an LU-factorization followed by forward- and backward substitution.

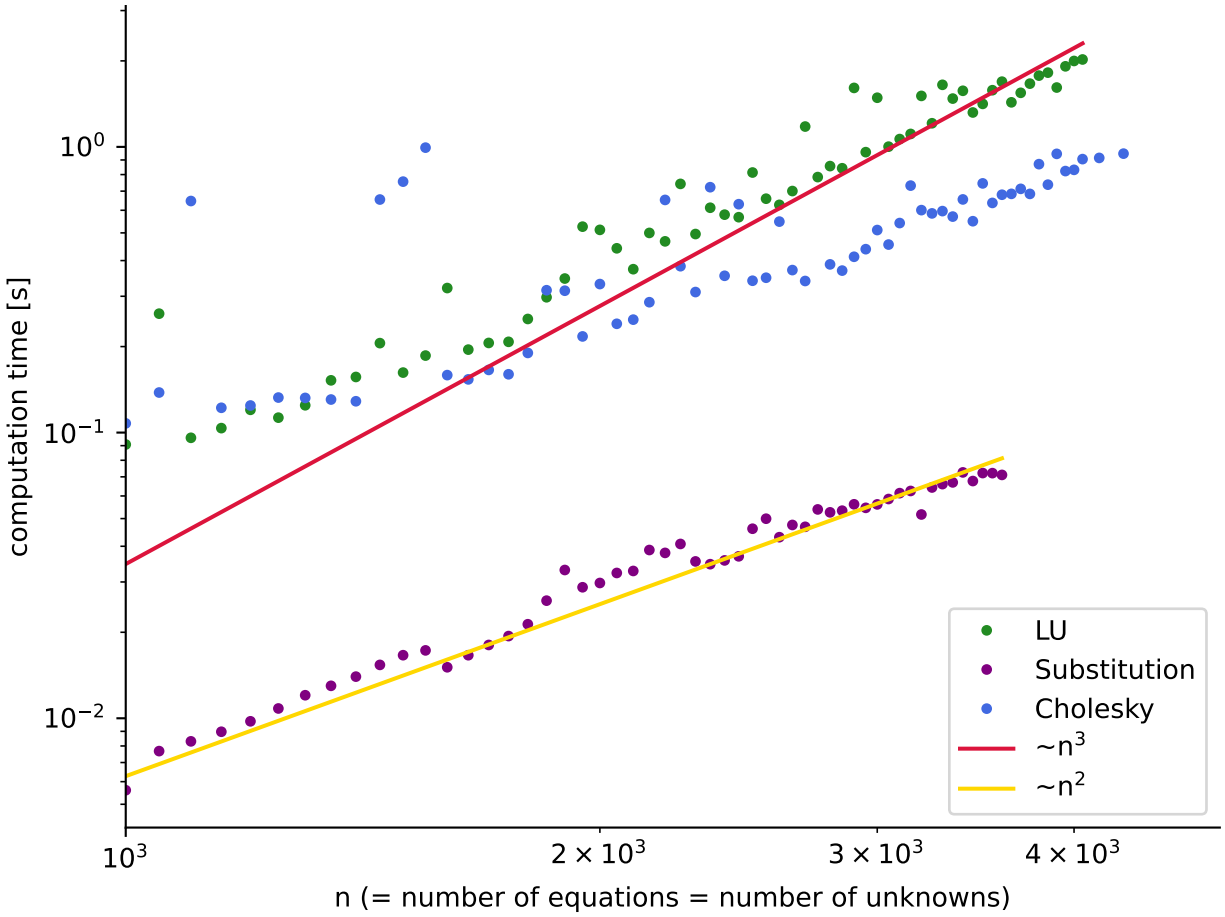
```
def compute_timings():
    """Compute timings of several linear algebra functions.
    WARNING: This code takes several minutes to run!
    """
    # Output arrays
    sizes = np.arange(1000, 5000, 50)
    timings_luf = np.zeros(sizes.shape)
    timings_tri = np.zeros(sizes.shape)
    timings_cho = np.zeros(sizes.shape)

    for isize, size in enumerate(sizes):
        # LU factorization of a random matrix
        A = np.random.random((size, size))
        start = timer()
        for _ in range(4):
            P, L, U = linalg.lu(A)
        timings_luf[isize] = timer() - start

        # Forward and backward subtitution takes
        start = timer()
        b = np.random.random(size)
        for _ in range(4):
            y = linalg.solve_triangular(L, b, lower=True)
            _x = linalg.solve_triangular(U, y, lower=False)
        timings_tri[isize] = timer() - start

        # Cholesky factorization
        C = np.dot(A, A.T) + np.identity(size)
        start = timer()
        for _ in range(4):
            L = linalg.cholesky(C)
        timings_cho[isize] = timer() - start

    return sizes, timings_luf, timings_tri, timings_cho


def filter_outliers(sizes, timings):
    mask = timings < timings.mean() * 1.2
    return sizes[mask], timings[mask]


sizes, timings_luf, timings_tri, timings_cho = compute_timings()
```

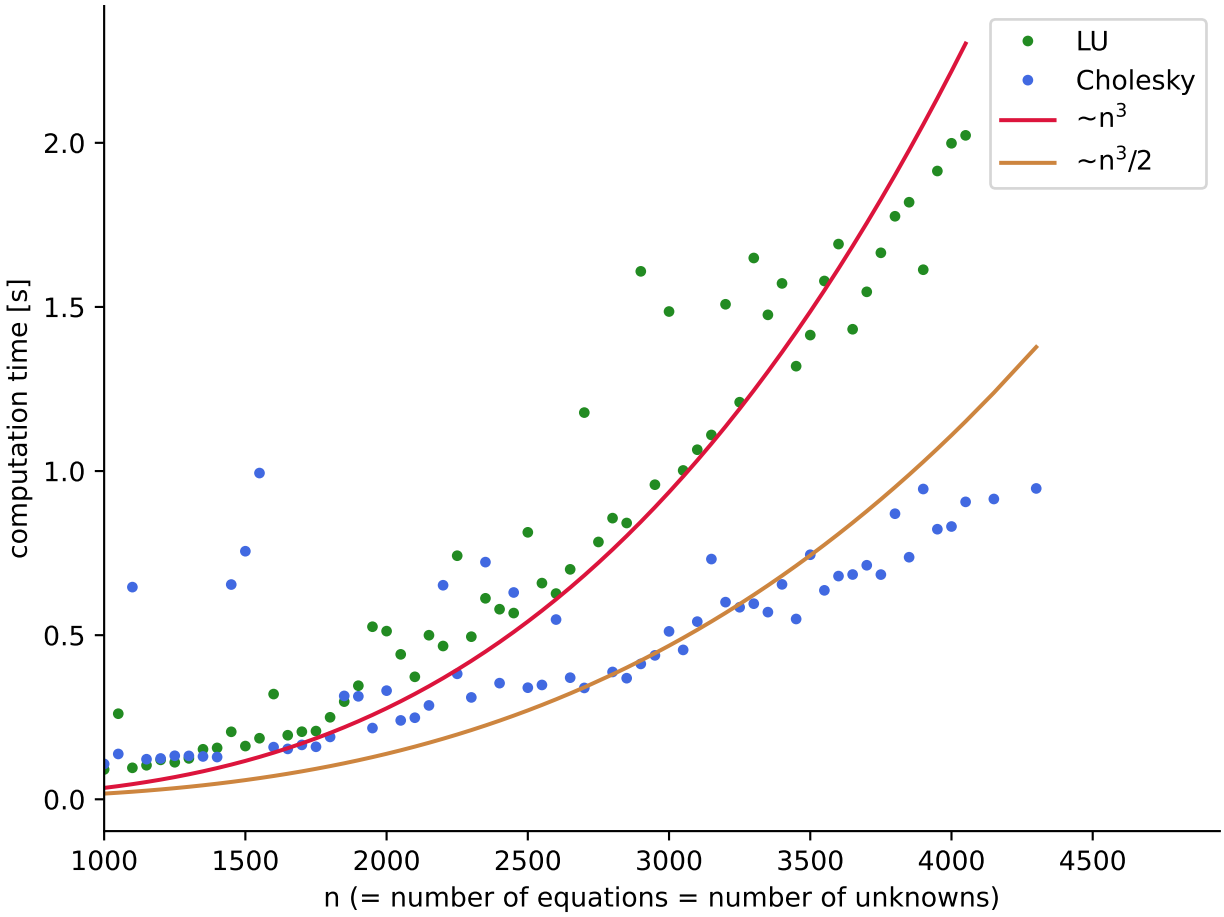
(continues on next page)

(continued from previous page)

```
sizes_luf, timings_luf = filter_outliers(sizes, timings_luf)
sizes_tri, timings_tri = filter_outliers(sizes, timings_tri)
sizes_cho, timings_cho = filter_outliers(sizes, timings_cho)
```

```
def plot_timings():
    # fit a polynomial to the data
    def cost_luf(x, a1):
        return a1 * x**3.0

    def cost_tri(x, a2):
        return a2 * x**2.0

    popt_luf = optimize.curve_fit(cost_luf, sizes_luf, timings_luf)[0]
    popt_tri = optimize.curve_fit(cost_tri, sizes_tri, timings_tri)[0]

    # plot the results of the previous operations
    plt.close("time1")
    fig, ax = plt.subplots(num="time1")
    ax.set_xlim(sizes.min(), sizes.max())
    ax.plot(sizes_luf, timings_luf, ".", label="LU", c="forestgreen")
    ax.plot(sizes_tri, timings_tri, ".", label="Substitution", c="purple")
    ax.plot(sizes_cho, timings_cho, ".", label="Cholesky", c="royalblue")
    ax.plot(sizes_luf, cost_luf(sizes_luf, popt_luf), label="~n$^3$", c=
 ↪"crimson")
    ax.plot(sizes_tri, cost_tri(sizes_tri, popt_tri), label="~n$^2$", c="gold
 ↪")
    ax.set_xscale("log")
    ax.set_yscale("log")
    ax.set_xlabel("n (= number of equations = number of unknowns)")
    ax.set_ylabel("computation time [s]")
    ax.legend()

    plt.close("time2")
    fig, ax = plt.subplots(num="time2")
    ax.set_xlim(sizes.min(), sizes.max())
    ax.plot(sizes_luf, timings_luf, ".", label="LU", c="forestgreen")
    ax.plot(sizes_cho, timings_cho, ".", label="Cholesky", c="royalblue")
    ax.plot(sizes_luf, cost_luf(sizes_luf, popt_luf), label="~n$^3$", c=
 ↪"crimson")
    ax.plot(
        sizes_cho, cost_luf(sizes_cho, popt_luf / 2), label="~n$^3 /2$", c=
 ↪"peru"
    )
    ax.set_xlabel("n (= number of equations = number of unknowns)")
    ax.set_ylabel("computation time [s]")
    ax.legend()


plot_timings()
```


computation time [s]
10^0
10^{-1}
10^{-2}
10^3
2×10^3
3×10^3
4×10^3
n (= number of equations = number of unknowns)
LU
Substitution
Cholesky
~n^3
~n^2



Warning: if you don't use sufficiently large matrices (>5000x5000) in the example above it seems that LU-factorization scales with O(n^2).

According to the scipy docs it implements the *GETRF routines from LAPACK which state that "The approximate number of floating-point operations for real flavors is (2/3)n^3 If m = n " in line with what we expect.

2.4 Sensitivity and Conditioning

We've seen how to find the solutions to a system of linear equations. Now let's consider how sensitive this solution is to small changes in the inputs. To do this we need to introduce a few concepts to extend the idea of an absolute value of a *scalar* to the **norm** of a *vector* or *matrix*.

2.4.1 Vector norms

There's more than one way to define a "size" of a vector, but all the **vector norms** we will consider here are **p-norms**, which for an integer $p > 0$ and an n-vector $\mathbf{x}$ are defined as

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} \|x_i\|^p \right)^{1/p}$$

Important cases:

- **1-norm** or **Manhattan norm** (because in two dimensions it corresponds to the distance between two points as measured in "city blocks":

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} \|x_i\|$$

- **2-norm** or **Euclidean norm** (because it corresponds to the usual notion of distance in Euclidean space):

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} \|x_i\|^2}$$

- **∞-norm** (the limit for $p \to \infty$):

$$\|\mathbf{x}\|_\infty = \max_{1 \le i \le n} \|x_i\|$$

> **Example**
>
> Consider the vector $\mathbf{x} = [-4, 3]^\mathrm{T}$
>
> $\|\mathbf{x}\|_1 = 7$
>
> $\|\mathbf{x}\|_2 = 5$
>
> $\|\mathbf{x}\|_\infty = 4$

The following cell calculates the p-norm for a vector **v**, using code presented on @introductiontoscientificpr4335's youtube channel

```
def pnorm(v, p):
    """function that calculates the p-norm for a vector v"""
    n = len(v)
    pn = 0
    for i in range(n):
        pn = pn + np.abs(v[i]) ** p
    pn = pn ** (1 / p)
    return pn
```

The function cannot compute the norm for $p > 11$ due to the factor $\frac{1}{p}$. It is better to use the SciPy function to calculate the norm.

```
v = np.array([-4, 3])
for p in range(1, 11):
    print(f"{p}-norm:{pnorm(v, p)}")
```

```
1-norm:7.0
2-norm:5.0
3-norm:4.497941445275415
4-norm:4.284572294953817
5-norm:4.174027662897746
6-norm:4.110704132575835
7-norm:4.072242319397026
8-norm:4.04799203437848
9-norm:4.032307299196485
10-norm:4.021974149822332
```

Note that the norm very quickly converges to the maximum element of the vector for large values of p.

In general, for any n-vector **x**:

$$\|\mathbf{x}\|_1 \ge \|\mathbf{x}\|_2 \ge \|\mathbf{x}\|_\infty$$

and

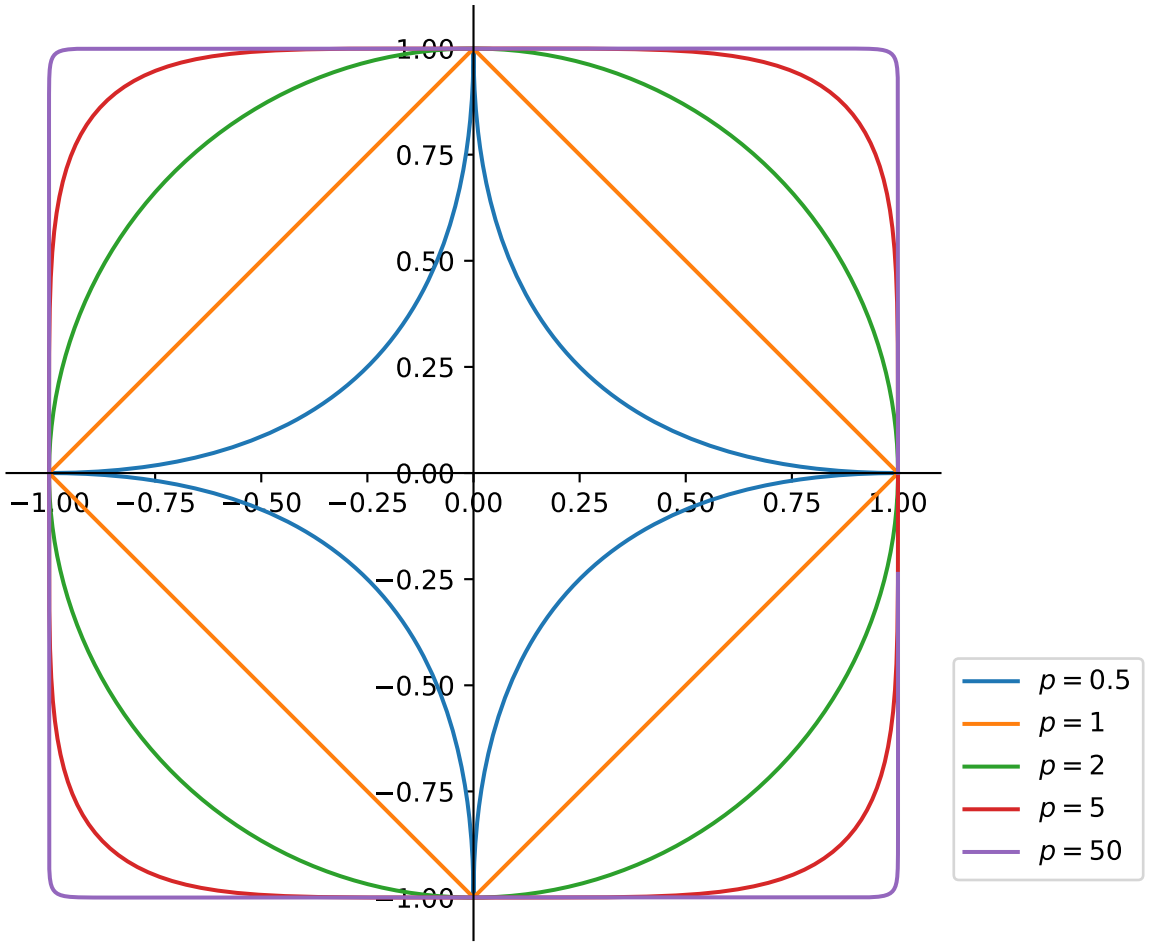
$$\|\mathbf{x}\|_1 \leq \sqrt{n}\|\mathbf{x}\|_2$$
$$\|\mathbf{x}\|_2 \leq \sqrt{n}\|\mathbf{x}\|_\infty$$
$$\|\mathbf{x}\|_1 \leq n\|\mathbf{x}\|_\infty$$

And for all p-norms, the following properties hold:

- $\|\mathbf{x}\| > 0$ if $\mathbf{x} \neq \mathbf{0}$
- $\|\gamma\mathbf{x}\| = |\gamma| \cdot \|\mathbf{x}\|$ for any scalar γ
- $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$ (triangle inequality)

"Circles with different norms", a.k.a. superellipses, are visualized below:

```
def unitball2d(pvals):
    plt.close("unitball2d")
    fig, ax = plt.subplots(figsize=(6, 5), num="unitball2d")
    theta = np.linspace(0, np.pi * 2, 370)
    for p in pvals:
        x0 = np.cos(theta)
        y0 = np.sin(theta)
        x1 = abs(x0) ** (2 / p) * np.sign(x0)
        y1 = abs(y0) ** (2 / p) * np.sign(y0)
        ax.plot(x1, y1, label=f"$p = {p}$")
    ax.set_aspect("equal")
    ax.spines["left"].set_position("center")
    ax.spines["right"].set_color("none")
    ax.spines["bottom"].set_position("center")
    ax.spines["top"].set_color("none")
    ax.legend(loc="lower left", bbox_to_anchor=(1.0, 0.05))


unitball2d(pvals=[0.5, 1, 2, 5, 50])
```



2.4.2 Matrix norms

To get a measure for the size of a matrix, we define a **matrix norm** using the definitions of vector norms.

The norm of an $m \times n$ matrix $\mathbf{A}$ is given by

$$\|\mathbf{A}\| = \max_{\mathbf{x} \neq \mathbf{0}} \frac{\|\mathbf{A}\mathbf{x}\|}{\|\mathbf{x}\|},$$

which corresponds to the maximum stretching the matrix does to a vector, as measured using a given vector norm.

Two cases that are easy to calculate are:

- $\|\mathbf{A}\|_1$, which corresponds the maximum absolute *column* sum of the matrix:

$$\|\mathbf{A}\|_1 = \max_j \sum_{i=1}^{m} \|a_{ij}\|$$

- $\|\mathbf{A}\|_\infty$, which corresponds the maximum absolute *row* sum of the matrix:

$$\|\mathbf{A}\|_\infty = \max_i \sum_{j=1}^{n} \|a_{ij}\|$$

Unfortunately, the Euclidean norm is not so easy to calculate as it corresponds to the square root of the largest eigenvalue of $\mathbf{A}^\mathsf{T}\mathbf{A}$ (see the chapter on eigenvalues).

Example

Consider the matrix

$$\mathbf{A} = \begin{bmatrix} 1 & -3 & 5 \\ 2 & 0 & 4 \\ 0 & -1 & 2 \end{bmatrix}$$

The norms corresponding to the maximum row and column sums are:

$$\|\mathbf{A}\|_1 = 11$$

$$\|\mathbf{A}\|_\infty = 9$$

Important properties of matrix norms:

- $\|\mathbf{A}\| > 0$ if $\mathbf{A} \neq \mathbf{0}$
- $\|\gamma\mathbf{A}\| = |\gamma| \cdot \|\mathbf{A}\|$, for any scalar γ
- $\|\mathbf{A} + \mathbf{B}\| \leq \|\mathbf{A}\| + \|\mathbf{B}\|$
- $\|\mathbf{AB}\| \leq \|\mathbf{A}\| \cdot \|\mathbf{B}\|$
- $\|\mathbf{Ax}\| \leq \|\mathbf{A}\| \cdot \|\mathbf{x}\|$, for any vector $\mathbf{x}$

2.4.3 Matrix condition number

The **condition number** of a nonsingular square matrix $\mathbf{A}$ with respect to a given matrix norm is given by

$$\text{cond}(\mathbf{A}) = \|\mathbf{A}\| \cdot \|\mathbf{A}^{-1}\|$$

By convention, the condition number of a singular matrix is ∞.

The condition number is a measure of how close a matrix is to being singular. The minimal condition number is 1 which only occurs for orthogonal matrices. A condition number close to 1 corresponds to a well-posed problem, whereas a very large condition number tells you that a solution of a linear system will change drastically for small changes in the input data.

Example

The inverse of the matrix from the previous example

$$\mathbf{A} = \begin{bmatrix} 1 & -3 & 5 \\ 2 & 0 & 4 \\ 0 & -1 & 2 \end{bmatrix}$$

is

$$\mathbf{A}^{-1} = \begin{bmatrix} 2/3 & 1/6 & -2 \\ -2/3 & 1/3 & 1 \\ -1/3 & 1/6 & 1 \end{bmatrix}$$

so that $\|\mathbf{A^{-1}}\|_1 = 4$ and $\|\mathbf{A^{-1}}\|_\infty = 17/6$

Thus the condition numbers are

$$\text{cond}_1(\mathbf{A}) = \|\mathbf{A}\|_1 \cdot \|\mathbf{A^{-1}}\|_1 = 11 \cdot 4 = 44$$

and

$$\text{cond}_\infty(\mathbf{A}) = \|\mathbf{A}\|_\infty \cdot \|\mathbf{A^{-1}}\|_\infty = 9 \cdot 17/6 = 25.5$$

Note that, because there is a strict order in the size of the different matrix norms (the 1-norm is always larger than the 2-norm, and so on) it doesn't matter too much which norm is used to estimate the condition number, as they will all give the same relative idea of how well-posed the problem underlying your linear system is.

2.4.4 Error estimation

Next to an estimate of how near a matrix is to being singular, the condition number also gives an estimate for the error on the solution of a system of linear equations.

Consider the non-singular system $\mathbf{Ax} = \mathbf{b}$ with solution $\mathbf{x}$. Additionally, let $\mathbf{x}'$ be the solution to the perturbed system $\mathbf{Ax}' = \mathbf{b} + \Delta\mathbf{b}$, and define the difference between both solutions $\mathbf{x}' - \mathbf{x}$ as $\Delta\mathbf{x}$.

This results in

$$\mathbf{Ax}' = \mathbf{A}(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{Ax} + \mathbf{A}\Delta\mathbf{x} = \mathbf{b} + \Delta\mathbf{b}$$

Consequently, $\mathbf{A}\Delta\mathbf{x} = \Delta\mathbf{b}$, and hence $\Delta\mathbf{x} = \mathbf{A}^{-1}\Delta\mathbf{b}$.

Taking norms, and using the properties listed above we find:

- $\|\mathbf{b}\| = \|\mathbf{Ax}\| \leq \|\mathbf{A}\| \cdot \|\mathbf{x}\|$ or $\|\mathbf{x}\| \geq \frac{\|\mathbf{b}\|}{\|\mathbf{A}\|}$
- $\|\Delta\mathbf{x}\| = \|\mathbf{A}^{-1}\Delta\mathbf{b}\| \leq \|\mathbf{A}^{-1}\| \cdot \|\Delta\mathbf{b}\|$

Combining both equalities gives

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \text{cond}(\mathbf{A})\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|}$$

The condition number thus acts as an "amplification factor" for the relative change in the solution with respect to a relative change in the right hand sided vector.

Example

Let's verify the expression above by revisiting a previous example and see how a perturbation in $\mathbf{b}$ affects the solution of the following linear system:

$$\mathbf{Ax} = \begin{bmatrix} 1 & 2 & 2 \\ 4 & 4 & 2 \\ 4 & 6 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \\ 10 \end{bmatrix} = \mathbf{b}$$

Remember that the solution $\mathbf{x}$ of this system was found to be equal to $[-1 \quad 3 \quad -1]^\intercal$.

If a perturbation $\Delta\mathbf{b} = [-1.8\ 0 \quad 0]^\intercal$ is applied to vector $\mathbf{b}$, the linear system to be solved becomes:

$$\mathbf{Ax}' = \begin{bmatrix} 1 & 2 & 2 \\ 4 & 4 & 2 \\ 4 & 6 & 4 \end{bmatrix} \begin{bmatrix} x'_1 \\ x'_2 \\ x'_3 \end{bmatrix} = \begin{bmatrix} 1.2 \\ 6 \\ 10 \end{bmatrix} = \mathbf{b}'$$

with a solution $\mathbf{x}' = [-2.8 \quad 6.6 \quad -4.6]^\intercal$. Hence,

$$\Delta\mathbf{x} = \mathbf{x}' - \mathbf{x} = \begin{bmatrix} -1.8 \\ 3.6 \\ -3.6 \end{bmatrix} \quad \Rightarrow \quad \frac{\|\Delta\mathbf{x}\|_1}{\|\mathbf{x}\|_1} = \frac{9}{5} = 1.8$$

> While
>
> $$\frac{\|\Delta\mathbf{b}\|_1}{\|\mathbf{b}\|_1} = \frac{1.8}{19} \approx 0.095$$
>
> Note that the relative change in $\mathbf{x}$ is larger that the relative change in $\mathbf{b}$ but that the following expression (with $\text{cond}(\mathbf{A}) = 60$) still holds:
>
> $$1.8 = \frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \text{cond}(\mathbf{A})\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|} = 60 \cdot \frac{1.8}{19} \approx 5.68,$$

A similar reasoning (which is left as an **exercise**) learns us that for deviations $\mathbf{E}$ in the matrix $\mathbf{A}$, such that $(\mathbf{A}+\mathbf{E})\mathbf{x}' = \mathbf{b}$, we find

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}'\|} \leq \text{cond}(\mathbf{A})\frac{\|\mathbf{E}\|}{\|\mathbf{A}\|}$$

Because the condition number cannot be smaller than 1 (and is only 1 for the identity matrix), relative errors in your inputs will always result in larger relative errors in the solution of your linear system.

Also note that, because the vector norm is dominated by the largest components of a vector, the relative error on the smallest components of the solution vector can be much larger than the relative error on the largest components.

2.4.5 Residual

An easy way to check the correctness of your solution to a system of linear equations is to fill it in the system and see whether the two sides of the equation are equal. The **residual** $\mathbf{r}$ of an approximate solution $\mathbf{x}'$ of the system $\mathbf{Ax} = \mathbf{b}$ is defined as

$$\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}'$$

In theory, for a nonsingular $\mathbf{A}$, the residual $\mathbf{r} = 0$ if and only if the error $\|\mathbf{x} - \mathbf{x}'\| = 0$. In practice, they are not necessarily small simultaneously.

If we multiply the system $\mathbf{Ax} = \mathbf{b}$ by an arbitrary number, the solution will remain the same, whereas the residual will be multiplied by the same number. Therefore, it only makes sense to work with a **relative residual**

$$\frac{\|\mathbf{r}\|}{(\|\mathbf{A}\| \cdot \|\mathbf{x}'\|)}$$

To relate the error to the residual, we observe

$$\|\Delta\mathbf{x}\| = \|\mathbf{x}' - \mathbf{x}\| = \|\mathbf{A}^{-1}(\mathbf{A}\mathbf{x}' - \mathbf{b})\| = \| - \mathbf{A}^{-1}\mathbf{r}\| \leq \|\mathbf{A}^{-1}\| \cdot \|\mathbf{r}\|$$

Dividing both sized by $\|\mathbf{x}'\|$ and filling in the definition of $\text{cond}(\mathbf{A})$, we find

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}'\|} \leq \text{cond}(\mathbf{A})\frac{\|\mathbf{r}\|}{\|\mathbf{A}\| \cdot \|\mathbf{x}'\|}$$

Therefore, a small residual implies a small relative error in the solution *only* if the matrix $\mathbf{A}$ has a small condition number.

2.5 Software

To work with systems of linear equations in python, we prefer the use of `scipy.linalg` which contains all the functions in `numpy.linalg` plus some other more advanced ones not contained in `numpy.linalg`.

The documentation for `scipy.linalg` can be found here:

https://docs.scipy.org/doc/scipy/reference/linalg.html

together with lots of useful tutorials here:

https://docs.scipy.org/doc/scipy/tutorial/linalg.html

2.5.1 Solving systems of linear equations

For the generic case, `scipy.linalg.solve` uses lapack gesv, which is based on LU factorization,

for more specific cases, symmetric, positive definite, banded, more specialized routines based e.g. on Cholesky factorization are used.

As an example, the next cell shows how to solve the system used in the examples above using `linalg.solve`. and how to perform an LU factorization using `linalg.lu`, which returns the permutation matrix **P**, used to pivot the matrix **A** and the lower and upper triangular matrices **L** and **U**, respectively.

These examples all use the same matrices as in the examples shown above.

```
A = np.array([[1, 2, 2], [4, 4, 2], [4, 6, 4]])
b = np.array([3, 6, 10])
x = linalg.solve(A, b)
print(x, "\n")
```

```
[-1.  3. -1.]
```

```
P, L, U = linalg.lu(A)
print("Matrix A =\n", A, "\n")
print("Permutation matrix P =\n", P, "\n")
print("Lower triangular matrix L =\n", L, "\n")
print("Upper triangular matrix U =\n", U, "\n")
print("Reconstructed matrix (P @ L @ U) =\n", P @ L @ U, "\n")
```

```
Matrix A =
 [[1 2 2]
 [4 4 2]
 [4 6 4]]

Permutation matrix P =
 [[0. 0. 1.]
 [1. 0. 0.]
 [0. 1. 0.]]

Lower triangular matrix L =
 [[1.   0.   0.  ]
 [1.   1.   0.  ]
 [0.25 0.5  1.  ]]

Upper triangular matrix U =
 [[4.  4.  2. ]
 [0.  2.  2. ]
 [0.  0.  0.5]]

Reconstructed matrix (P @ L @ U) =
 [[1. 2. 2.]
 [4. 4. 2.]
 [4. 6. 4.]]
```

Once you have the LU factorization, you can use it to efficiently solve your linear system (especially if you have multiple **b** vectors for which you want to solve it) using `linalg.lu_solve`.

```
A = np.array([[1, 2, 2], [4, 4, 2], [4, 6, 4]])
b = np.array([3, 6, 10])
```

(continues on next page)

(continued from previous page)

```
lu, piv = linalg.lu_factor(A)
x = linalg.lu_solve((lu, piv), b)
print(x)
```

```
[-1.  3. -1.]
```

The Cholesky factorization can be calculated with `linalg.cholesky`.

```
a = np.array([[4, 2, 6], [2, 2, 5], [6, 5, 22]])
L = linalg.cholesky(a, lower=True)

print(L)
```

```
[[2. 0. 0.]
 [1. 1. 0.]
 [3. 2. 3.]]
```

```
# There are also functions to calculate the inverse of a matrix and its norm

A = np.array([[1.0, -3.0, 5.0], [2.0, 0.0, 4.0], [0.0, -1.0, 2.0]])

print("The 1-norm of A equals: ", linalg.norm(A, 1))
print("The infty-norm of A equals: ", linalg.norm(A, np.inf))

B = linalg.inv(A)

print("\n\nThe inverse of A is: \n", B, "\n\n")
print("The 1-norm of A^-1 equals: ", linalg.norm(B, 1))
print("The infty-norm of A^-1 equals: ", linalg.norm(B, np.inf))

print("\n\nThe condition number of A, thus is:")
print(linalg.norm(A, 1) * linalg.norm(B, 1), " for the 1-norm")
print(linalg.norm(A, np.inf) * linalg.norm(B, np.inf), " for the infty-norm")

# TODO add condition number
```

```
The 1-norm of A equals:  11.0
The infty-norm of A equals:  9.0


The inverse of A is:
 [[ 0.66666667  0.16666667 -2.        ]
 [-0.66666667  0.33333333  1.        ]
 [-0.33333333  0.16666667  1.        ]]


The 1-norm of A^-1 equals:  4.0
The infty-norm of A^-1 equals:  2.833333333333333


The condition number of A, thus is:
44.0  for the 1-norm
25.499999999999996  for the infty-norm
```

CHAPTER 3

Linear Least Squares

```
import matplotlib.pyplot as plt
import numpy as np
from scipy import linalg
```

3.1 Introduction

In the previous chapter we saw methods to solve systems of linear equations with the same amount of unknowns as equations. For a nonsingular problem, there was only one solution, for which each of the equations was exactly satisfied.

However, in a scientific experiment, you'll often have far more measurement data points than unknown variables in the function you want to fit to your data. In the terminology of the previous chapter, you have a system

$$\mathbf{Ax} = \mathbf{b}$$

In which $\mathbf{A}$ no longer is a square matrix but an $m \times n$ matrix with $m > n$. We call this an *overdetermined* problem. Typically, all measurement data also contain a certain amount of noise, such that there no longer exists a solution that exactly satisfies all equations.

Instead, we want to model the data as closely as possible by minimizing the norm of the residual $\mathbf{r} = \mathbf{b} - \mathbf{Ax}$ as a function of $\mathbf{x}$.

In principle, any norm can be used for this purpose, but the best choice is the Euclidean norm, which also gives the method of least squares its name: the solution is the vector $\mathbf{x}$ that minimizes the sum of squares of differences between the components of the left and right sides of the linear system.

To reflect the lack of an exact equality, we write such a problem as $\mathbf{Ax} \cong \mathbf{b}$.

> **Example: curve fitting**
>
> Given data points (t_i, y_i), $i = 1, \ldots, m$ we wish to find the set of parameters $\mathbf{x}$ that gives the best fit to the data by a model function $f(t, \mathbf{x})$ in the sense:
>
> $$\min_{\mathbf{x}} \sum_{i=1}^{m} (y_i - f(t_i, \mathbf{x}))^2$$

A data-fitting problem is *linear* if the function f is linear in the components of the parameter vector $\mathbf{x}$. In this chapter, we will limit ourselves to *linear* least squares, and leave *nonlinear* least squares for the chapter on *optimization*.

For instance, when fitting a quadratic polynomial (3 parameters) to a dataset containing 6 data points $(t_1, y_1), \ldots, (t_6, y_6)$ the problem has the form

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} 1 & t_1 & t_1^2 \\ 1 & t_2 & t_2^2 \\ 1 & t_3 & t_3^2 \\ 1 & t_4 & t_4^2 \\ 1 & t_5 & t_5^2 \\ 1 & t_6 & t_6^2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \cong \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} = \mathbf{b}$$

Such a matrix, whose columns are successive powers of some independent variable is called a *Vandermonde matrix*.

Example: projectile trajectory

Suppose we throw a ball in the air and record the height (y) of the ball as a function of time (t). Thanks to the laws of kinematics we know that the following relation holds:

$$f(t) = y(t) = y_0 + v_0 t + \frac{1}{2} a t^2$$

From the experiment we obtain the following 10 data points:

t	0	1	2	3	4	5	6	7	8	9	10
y	2.9	4.8	7.1	7.7	9.4	9.6	10.2	8.3	9.0	6.6	4.1

This corresponds to the linear system

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 4 & 16 \\ 1 & 5 & 25 \\ \vdots & \vdots & \vdots \\ 1 & 10 & 100 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \cong \begin{bmatrix} 2.9 \\ 4.8 \\ 7.1 \\ 7.7 \\ 9.4 \\ 9.6 \\ \vdots \\ 4.1 \end{bmatrix} = \mathbf{b}$$

The solution to this system (which we'll learn how to find in the remainder of this chapter) is

$$\mathbf{x} \approx [2.60489151 \quad 2.65037296 \quad -0.2460373]^\intercal,$$

corresponding to the polynomial

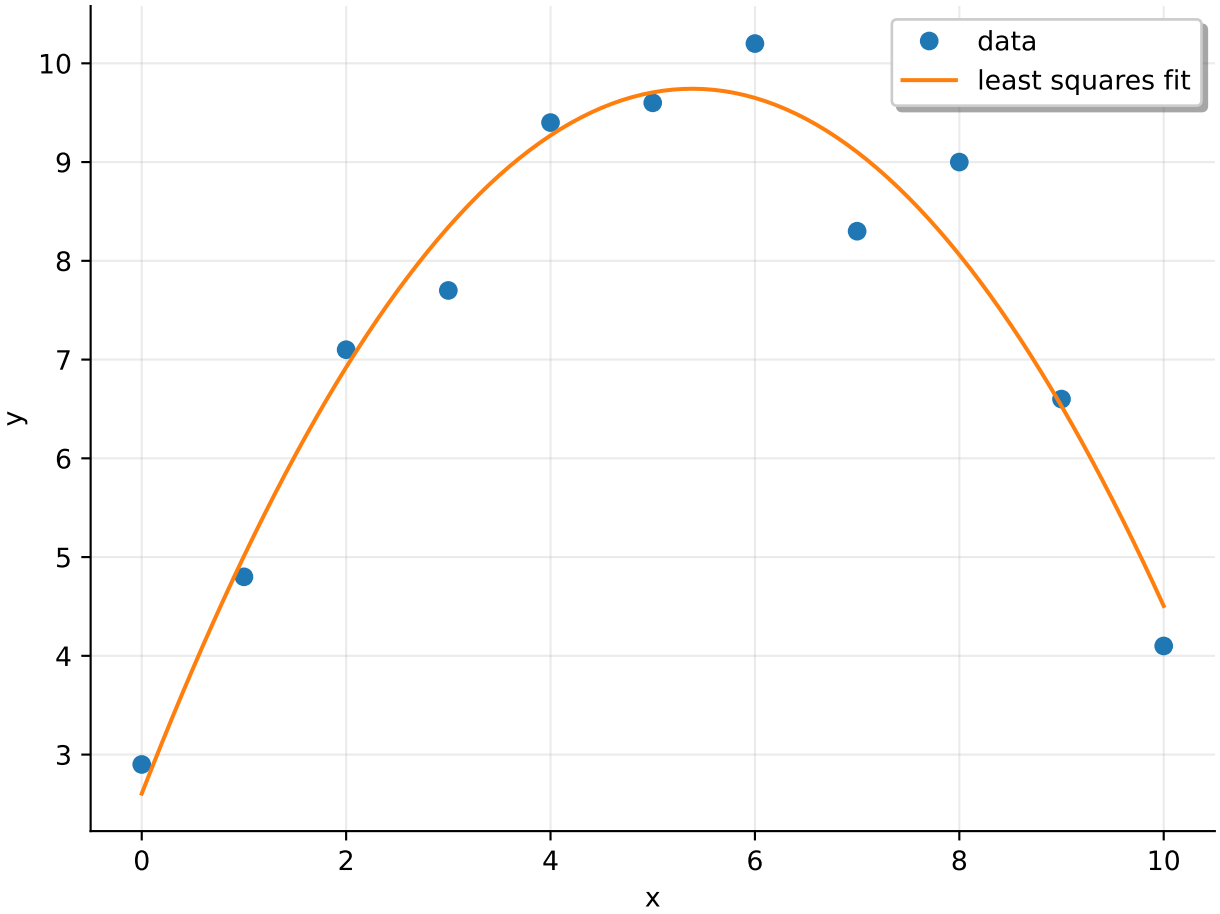
$$f(t) = 2.60 + 2.65t - 0.25t^2$$

from which it follows that the experiment was apparently performed on Pluto ($g = 0.5\,\mathrm{m/s^2}$) and not on earth ($g = 9.8\,\mathrm{m/s^2}$).

Although we still don't know how the computer solves this kind of problem, we can already illustrate how you can perform this kind of least squares fitting using `scipy`. We illustrate this using the same example:

```
def lstsq_example1():
    x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
    y = np.array([2.9, 4.8, 7.1, 7.7, 9.4, 9.6, 10.2, 8.3, 9.0, 6.6, 4.1])

    # We want to fit a quadratic polynomial of the form
    # y = a + b*x +c*x**2 to this data.
    # We first form the "design matrix" M,
    # with a constant column of 1s, a column containing x and a column x**2:
    M = x[:, np.newaxis] ** [0, 1, 2]
    print(M)

    # We want to find the least-squares solution to M.dot(p) = y,
    # where p is a vector with length 3 that holds the parameters a and b an␣
 ↪c.
    p, res, rnk, s = linalg.lstsq(M, y)

    print(p)

    plt.close("lstsq-example-1")
    fig, ax = plt.subplots(num="lstsq-example-1")
    ax.plot(x, y, "o", label="data")
    xx = np.linspace(0, 10, 101)
    yy = p[0] + p[1] * xx + p[2] * xx**2
    ax.plot(xx, yy, label="least squares fit")
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.legend(framealpha=1, shadow=True)
    ax.grid(alpha=0.25)


lstsq_example1()
```

```
[[  1   0   0]
 [  1   1   1]
 [  1   2   4]
 [  1   3   9]
 [  1   4  16]
 [  1   5  25]
 [  1   6  36]
 [  1   7  49]
 [  1   8  64]
 [  1   9  81]
 [  1  10 100]]
[ 2.6048951   2.65037296 -0.2460373 ]
```



3.2 Normal Equations

We wish to minimize the squared Euclidean norm of the residual vector

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

We define an objective function

$$\phi(\mathbf{x}) = \|\mathbf{r}\|_2^2 = \mathbf{r}^\intercal\mathbf{r} = (\mathbf{b} - \mathbf{A}\mathbf{x})^\intercal(\mathbf{b} - \mathbf{A}\mathbf{x}) = \mathbf{b}^\intercal\mathbf{b} - \mathbf{x}^\intercal\mathbf{A}^\intercal\mathbf{b} - \mathbf{b}^\intercal\mathbf{A}\mathbf{x} + \mathbf{x}^\intercal\mathbf{A}^\intercal\mathbf{A}\mathbf{x}$$

To minimize this function, we want to find a point which satisfies $\nabla\phi(\mathbf{x}) = \mathbf{0}$ which is true for

$$\mathbf{0} = \nabla\phi(\mathbf{x}) = 2\mathbf{A}^\intercal\mathbf{A}\mathbf{x} - 2\mathbf{A}^\intercal\mathbf{b}$$

> Where we used the identity
>
> - $(\mathbf{B}\mathbf{A})^\intercal = \mathbf{A}^\intercal\mathbf{B}^\intercal$
>
> and
>
> - $\nabla(\mathbf{x}^\intercal\mathbf{A}^\intercal\mathbf{A}\mathbf{x}) = 2\mathbf{A}^\intercal\mathbf{A}\mathbf{x}$
> - $\nabla(\mathbf{b}^\intercal\mathbf{A}\mathbf{x}) = \mathbf{A}^\intercal\mathbf{b}$
> - $\nabla(\mathbf{x}^\intercal\mathbf{A}^\intercal\mathbf{b}) = \mathbf{A}^\intercal\mathbf{b}$

- $\nabla(\mathbf{b}^\intercal\mathbf{b}) = 0$

In other words, to minimize $\mathbf{x}$ for ϕ we need to satisfy the $n \times n$ symmetric linear system

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

This system is known as the **normal equations**. The solution is unique if and only if the columns of A are linearly independent, i.e., $\text{rank}(\mathbf{A}) = n$. In this case, the matrix $\mathbf{A}^\intercal\mathbf{A}$ is positive definite and symmetric, and the linear system can be solved efficiently with Cholesky factorization.

When some columns of $\mathbf{A}$ are linearly dependent, a manifold of solutions exists, i.e. $\text{rank}(\mathbf{A}) < n$. In this case, the matrix $\mathbf{A}^\intercal\mathbf{A}$ is positive semi-definite (some eigenvalues are zero) and the minimum of ϕ becomes degenerate.

3.3 Problem Transformations

There are several methods to solve a least-square system, but they all amount to the same strategy: to transform it into a square (ultimately triangular) linear system for which we can compute the exact solution using the methods we saw earlier.

Because $\text{cond}(\mathbf{A}^\intercal\mathbf{A}) = [\text{cond}(\mathbf{A})]^2$, solving the normal equations is not a stable algorithm to solve the least squares problem.

Therefore, we'll see methods to obtain the solution without explicitly calculating $\mathbf{A}^\intercal\mathbf{A}$.

To this end, we want to transform our linear system to a form that is simpler to solve (we'll see below that a triangular system is a suitable target). However, reducing a matrix using Gaussian Elimination is not an option in this context, as it does not preserve the Euclidean norm and therefore also not the least square solution.

3.3.1 Orthogonal transformations

We need a type of linear transformation which does preserve the Euclidean norm.

A square real matrix $\mathbf{Q}$ is *orthogonal* if its columns are *orthonormal*, meaning that $\mathbf{Q}^\intercal\mathbf{Q} = \mathbf{I}$.

Such an **orthogonal transformation** $\mathbf{Q}$ preserves the Euclidean norm of any vector $\mathbf{v}$:

$$\|\mathbf{Q}\mathbf{v}\|_2^2 = (\mathbf{Q}\mathbf{v})^\intercal\mathbf{Q}\mathbf{v} = \mathbf{v}^\intercal\mathbf{Q}^\intercal\mathbf{Q}\mathbf{v} = \mathbf{v}^\intercal\mathbf{v} = \|\mathbf{v}\|_2^2$$

Orthogonal matrices are very useful in numerical computations because their norm-preserving property means that they do not amplify errors, and can e.g. be used to solve square linear systems without requiring pivoting for numerical stability.

On the other hand, orthogonalization methods are computationally more expensive than methods based on Gaussian elimination, so their superior numerical properties come at a price that may or may not be worthwhile.

3.3.2 Triangular least squares problems

Before going into detail on how we'll transform our system, let's see why triangular systems are a suitable target for our transformation.

Consider a least squares system having an upper triangular matrix. In the overdetermined case $m > n$, such a problem has the form

$$\begin{bmatrix}\mathbf{R}\\ \mathbf{O}\end{bmatrix}\mathbf{x} \cong \begin{bmatrix}\mathbf{c}_1\\ \mathbf{c}_2\end{bmatrix}$$

with $\mathbf{R}$ an $n \times n$ upper triangular matrix and $\mathbf{O}$ a $(m-n) \times n$ null matrix.

The least squares residual is given by

$$\|\mathbf{r}\|_2^2 = \|\mathbf{c}_1 - \mathbf{R}\mathbf{x}\|_2^2 + \|\mathbf{c}_2\|_2^2$$

If we solve the triangular system $\mathbf{Rx} = \mathbf{c}_1$ (which can easily be achieved with back-substitution) we have found the least squares solution $\mathbf{x}$ and we can conclude that the minimum sum of squares is

$$\|\mathbf{r}\|_2^2 = \|\mathbf{c}_2\|_2^2$$

3.3.3 QR- Factorization

Transforming a system to triangular form is accomplished by **QR factorization**, which, for an $m \times n$ matrix $\mathbf{A}$ with $m > n$ has the form

$$\mathbf{A} = \mathbf{Q} \begin{bmatrix} \mathbf{R} \\ \mathbf{O} \end{bmatrix}$$

where $\mathbf{Q}$ is an $m \times m$ orthogonal matrix and $\mathbf{R}$ is an $n \times n$ upper triangular matrix.

The transformed right-hand side then reads

$$\mathbf{Q}^\intercal \mathbf{b} = \begin{bmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \end{bmatrix}$$

The residual equals

$$\|\mathbf{r}\|_2^2 = \|\mathbf{b} - \mathbf{Ax}\|_2^2 = \|\mathbf{b} - \mathbf{Q} \begin{bmatrix} \mathbf{R} \\ \mathbf{O} \end{bmatrix} \mathbf{x}\|_2^2 = \|\mathbf{Q}^\intercal \mathbf{b} - \begin{bmatrix} \mathbf{R} \\ \mathbf{O} \end{bmatrix} \mathbf{x}\|_2^2 = \|\mathbf{c}_1 - \mathbf{Rx}\|_2^2 + \|\mathbf{c}_2\|_2^2$$

We saw before that the solution to $\mathbf{Rx} = \mathbf{c}_1$, then gives the least squares solution $\mathbf{x}$ for the original problem.

QR factorization can be achieved with different methods, but in contrast to the LU factorization we saw earlier, we need to use transformations that preserve the Euclidean norm, i.e. orthogonal transformations. A few common choices are:

- Householder transformations
- Givens transformations
- Gram-Schmidt orthogonalization

We'll only consider Householder transformations because in practice this is the most effective and most used method.

3.3.4 Householder transformations

We seek an orthogonal transformation which annihilates targeted components of a vector. This is achieved by the Householder matrix

$$\mathbf{H} = \mathbf{I} - 2\frac{\mathbf{v}\mathbf{v}^\intercal}{\mathbf{v}^\intercal \mathbf{v}}$$

with $\mathbf{v}$ a nonzero vector. It can be shown that $\mathbf{H} = \mathbf{H}^{-1} = \mathbf{H}^\intercal$, which means that $\mathbf{H}$ is orthogonal and symmetric.

Annihilating all but the first component of a vector

We want to determine $\mathbf{v}$ such that it annihilates all the components of a vector $\mathbf{a}$ except the first, i.e.

$$\mathbf{Ha} = \begin{bmatrix} \alpha \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \alpha \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \alpha \mathbf{e}_1$$

Using the definition of $\mathbf{H}$ we find

$$\alpha \mathbf{e}_1 = \mathbf{Ha} = \left(\mathbf{I} - 2\frac{\mathbf{v}\mathbf{v}^\intercal}{\mathbf{v}^\intercal \mathbf{v}} \right) \mathbf{a} = \mathbf{a} - 2\mathbf{v}\frac{\mathbf{v}^\intercal \mathbf{a}}{\mathbf{v}^\intercal \mathbf{v}}$$

and thus

$$\mathbf{v} = (\mathbf{a} - \alpha\mathbf{e}_1)\frac{\mathbf{v}^\intercal\mathbf{v}}{2\mathbf{v}^\intercal\mathbf{a}}$$

The scalar factor is irrelevant as it cancels out in the expression for $\mathbf{H}$, so we find

$$\mathbf{v} = (\mathbf{a} - \alpha\mathbf{e}_1)$$

To preserve the norm and avoid cancellation

$$\alpha = -\text{sign}(a_1)\|\mathbf{a}\|_2$$

Annihilating all but the first k components of a vector

If we split up a given m-vector $\mathbf{a}$ as

$$\mathbf{a} = \begin{bmatrix}\mathbf{a}_1\\ \mathbf{a}_2\end{bmatrix}$$

where $\mathbf{a}_1$ is a $(k-1)$-vector with $1 \leq k < m$.

If we then take the householder vector to be

$$\mathbf{v} = \begin{bmatrix}\mathbf{0}\\ \mathbf{a}_2\end{bmatrix} - \alpha\mathbf{e}_k$$

where $\alpha = -\text{sign}(a_k)\|\mathbf{a}_2\|_2$, then the resulting Householder transformation annihilates the last $m-k$ components of $\mathbf{a}$.

QR factorization using householder transformations

By sequentially performing this transformation for all the columns from left to right of a matrix $\mathbf{A}$, we can get the desired upper triangular matrix:

$$\mathbf{H}_n \ldots \mathbf{H}_1\mathbf{A} = \begin{bmatrix}\mathbf{R}\\ \mathbf{O}\end{bmatrix}$$

The product of orthogonal householder transformations is itself an orthogonal matrix, which we define as

$$\mathbf{Q}^\intercal = \mathbf{H}_n \ldots \mathbf{H}_1 \qquad \text{or, equivalently} \qquad \mathbf{Q} = \mathbf{H}_n^\intercal \ldots \mathbf{H}_1^\intercal$$

Such that

$$\mathbf{A} = \mathbf{Q}\begin{bmatrix}\mathbf{R}\\ \mathbf{O}\end{bmatrix}$$

which shows that we have indeed calculated the QR factorization of $\mathbf{A}$.

To solve the least squares system $\mathbf{Ax} \cong \mathbf{b}$, we solve the equivalent system

$$\begin{bmatrix}\mathbf{R}\\ \mathbf{O}\end{bmatrix}\mathbf{x} \cong \mathbf{Q}^\intercal\mathbf{b} = \begin{bmatrix}\mathbf{c}_1\\ \mathbf{c}_2\end{bmatrix}$$

3.3.5 Example of householder transformation

Consider the following situation where Archimedes has to determine the density of gold (G), silver (S) and the crown © of the king to determine whether the crown is made from pure gold or not. Archimedes cannot damage the crown but the king is kind enough to give him two clumps, one of pure gold and one of pure silver, with the exact same mass as the crown. After sitting in his bath, Archimedes decides to submerge each object in water and determine their respective volumes by measuring how much the water rises. From this he can calculate the density $\rho = m/V$ and gets:

$$\rho_G = 19.3\,\text{kg/m}^3$$
$$\rho_S = 10.5\,\text{kg/m}^3$$
$$\rho_C = 16.6\,\text{kg/m}^3$$

To confirm these measurements, Archimedes also fetches a scale. He realizes that, although the scale is perfectly balanced in air (all the objects have the same mass m and exert the same force $F_N = mg$ on the scale), this will no longer be the case when the scale and the objects are submerged in water. Because of the difference in density, the three objects will have different volumes and they will displace a different amount of water. Because the bouyant force is equal to the weight of the displaced fluid the force exerted on the weighing scale by the objects under water will be different and equal to

$$F_N = mg - \rho_w V_{\text{obj}} g$$

where the last term represents the buoyant force with ρ_w the density of water and V_{obj} the volume of the object. By comparing the objects and recording the apparent weight difference under water ($\Delta m_{1,2} g = F_{N,1} - F_{N,2}$), Archimedes gets:

$$\begin{aligned}
\rho_S - \rho_G &= -8.7\text{kg/m}^3 \\
\rho_C - \rho_G &= -2.6\text{kg/m}^3 \\
\rho_C - \rho_S &= 6.2\text{kg/m}^3
\end{aligned}$$

The system we want to solve thus reads

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ -1 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} \rho_G \\ \rho_S \\ \rho_C \end{bmatrix} \cong \begin{bmatrix} 19.3 \\ 10.5 \\ 16.6 \\ -8.7 \\ -2.6 \\ 6.2 \end{bmatrix} = \mathbf{b}$$

Below, we show how this is solved sequentially by performing the householder transform on each of the columns successively, and finally to solve the upper triangular system to find our best estimate for $\mathbf{x}$ and the corresponding residual $\mathbf{r}$.

```
A = np.array([(1.0, 0, 0), (0, 1, 0), (0, 0, 1), (-1, 1, 0), (-1, 0, 1), (0, -
↪1, 1)])
b = np.array([[19.3, 10.5, 16.6, -8.7, -2.6, 6.2]]).T
```

```
e1 = np.zeros((len(A), 1))
e1[0][0] = 1.0
a1 = A[:, [0]]
alfa = np.linalg.norm(a1) * (-1) * np.sign(a1[0])
v1 = a1 - alfa * e1
print(
    "e1=\n",
    e1,
    "\n\na1=\n",
    a1,
    "\n\nalfa=\n",
    alfa,
    "\n\nv1=\n",
    v1,
    "\n\n",
)
```

```
e1=
 [[1.]
 [0.]
 [0.]
```

(continues on next page)

(continued from previous page)

```
 [0.]
 [0.]
 [0.]]

a1=
 [[ 1.]
 [ 0.]
 [ 0.]
 [-1.]
 [-1.]
 [ 0.]]

alfa=
 [-1.73205081]

v1=
 [[ 2.73205081]
 [ 0.        ]
 [ 0.        ]
 [-1.        ]
 [-1.        ]
 [ 0.        ]]
```

```
def householder(v, A):
    Q = np.zeros(np.shape(A))
    for i in range(len(A[0])):
        B = A[:, [i]] - 2.0 * np.transpose(v) @ A[:, [i]] / (np.transpose(v)␣
 ↪@ v) * v
        for j in range(len(A)):
            Q[j, [i]] = B[j]
    return Q
```

```
A = householder(v1, A)
b = householder(v1, b)
print("A =\n", A, "\n\nb=\n", b)
```

```
A =
 [[-1.73205081  0.57735027  0.57735027]
 [ 0.          1.          0.        ]
 [ 0.          0.          1.        ]
 [ 0.          0.78867513 -0.21132487]
 [ 0.         -0.21132487  0.78867513]
 [ 0.         -1.          1.        ]]

b=
 [[-17.66691824]
 [ 10.5       ]
 [ 16.6       ]
 [  4.83083117]
 [ 10.93083117]
 [  6.2       ]]
```

```
e2 = np.zeros((len(A), 1))
e2[1][0] = 1
a2 = A[:, [1]]
a2[0] = 0
alfa = np.linalg.norm(A[1:, [1]]) * (-1.0) * np.sign(a2[1])
v2 = a2 - alfa * e2
print("e2=\n", e2, "\n\na2=\n", a2, "\n\nalfa=\n", alfa, "\n\nv2=\n", v2, "\n\
↪n")
```

```
e2=
 [[0.]
 [1.]
 [0.]
 [0.]
 [0.]
 [0.]]

a2=
 [[ 0.        ]
 [ 1.        ]
 [ 0.        ]
 [ 0.78867513]
 [-0.21132487]
 [-1.        ]]

alfa=
 [-1.63299316]

v2=
 [[ 0.        ]
 [ 2.63299316]
 [ 0.        ]
 [ 0.78867513]
 [-0.21132487]
 [-1.        ]]
```

```
A = householder(v2, A)
b = householder(v2, b)
print("A=\n", A, "\n\nb=\n", b)
```

```
A=
 [[-1.73205081  0.57735027  0.57735027]
 [ 0.         -1.63299316  0.81649658]
 [ 0.          0.          1.        ]
 [ 0.          0.          0.03324491]
 [ 0.          0.          0.72314286]
 [ 0.          0.          0.68989795]]

b=
 [[-17.66691824]
 [ -3.55176013]
 [ 16.6       ]
 [  0.62182905]
 [ 12.05862989]
```

(continues on next page)

(continued from previous page)

```
 [ 11.53680084]]
```

```
e3 = np.zeros((len(A), 1))
e3[2][0] = 1
a3 = A[:, [2]]
a3[0] = 0
a3[1] = 0
alfa = np.linalg.norm(A[2:, [2]]) * (-1) * np.sign(a3[2])
v3 = a3 - alfa * e3
print("e3=\n", e3, "\n\na3=\n", a3, "\n\nalfa=\n", alfa, "\n\nv3=\n", v3, "\n\
 ↪n")
```

```
e3=
 [[0.]
 [0.]
 [1.]
 [0.]
 [0.]
 [0.]]

a3=
 [[0.        ]
 [0.        ]
 [1.        ]
 [0.03324491]
 [0.72314286]
 [0.68989795]]

alfa=
 [-1.41421356]

v3=
 [[0.        ]
 [0.        ]
 [2.41421356]
 [0.03324491]
 [0.72314286]
 [0.68989795]]
```

```
A = householder(v3, A)
b = householder(v3, b)
print("A=\n", A, "\n\nb=\n", b)
```

```
A=
 [[-1.73205081e+00  5.77350269e-01  5.77350269e-01]
 [ 0.00000000e+00 -1.63299316e+00  8.16496581e-01]
 [ 0.00000000e+00  0.00000000e+00 -1.41421356e+00]
 [ 0.00000000e+00  0.00000000e+00  6.93889390e-18]
 [ 0.00000000e+00  0.00000000e+00  1.11022302e-16]
 [ 0.00000000e+00  0.00000000e+00  1.11022302e-16]]

b=
 [[-17.66691824]
```

(continues on next page)

(continued from previous page)

```
 [ -3.55176013]
 [-23.54665581]
 [  0.06898979]
 [  0.03327805]
 [  0.06428825]]
```

```
R = A[:3, :]
c1 = b[:3]
c2 = b[3:]
print("R=\n", R, "\n\nc1=\n", c1, "\n\nc2=", c2)
```

```
R=
 [[-1.73205081  0.57735027  0.57735027]
 [ 0.         -1.63299316  0.81649658]
 [ 0.          0.         -1.41421356]]

c1=
 [[-17.66691824]
 [ -3.55176013]
 [-23.54665581]]

c2= [[0.06898979]
 [0.03327805]
 [0.06428825]]
```

Which gives us the problem in a form which is easy to solve:

$$\begin{bmatrix}\mathbf{R}\\ \mathbf{O}\end{bmatrix}\mathbf{x} \cong \mathbf{Q}^{\intercal}\mathbf{b} = \begin{bmatrix}\mathbf{c}_1\\ \mathbf{c}_2\end{bmatrix}$$

Let's solve it with `scipy` using the methods seen in the previous chapter.

```
x = linalg.solve(R, c1)
print("x=\n", x, "\n\nres=\n", np.linalg.norm(c2))
```

```
x=
 [[19.25]
 [10.5 ]
 [16.65]]

res=
 0.10000000000000234
```

It is clear that the density of the crown is lower than the density of gold and that the jeweler made the crown out of a mixture of gold and the much cheaper silver.

Now let's perform the same QR-decomposition immediately using `scipy`, and solve the resulting system to arrive at the same solution.

```
A = np.array([(1.0, 0, 0), (0, 1, 0), (0, 0, 1), (-1, 1, 0), (-1, 0, 1), (0, -
 ↪1, 1)])
b = np.array([[19.3, 10.5, 16.6, -8.7, -2.6, 6.2]]).T
Q, R = linalg.qr(A)
print("R=\n", R, "\n\nQ=\n", Q, "\n\n")
```

(continues on next page)

(continued from previous page)

```
x = linalg.solve(R[:3], (Q.T @ b)[:3])
print("x=\n", x, "\n\nres=\n", np.linalg.norm((Q.T @ b)[3:]))
```

```
R=
 [[-1.73205081  0.57735027  0.57735027]
 [ 0.         -1.63299316  0.81649658]
 [ 0.          0.         -1.41421356]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]
 [ 0.          0.          0.        ]]

Q=
 [[-5.77350269e-01 -2.04124145e-01 -3.53553391e-01  5.11339220e-01
   4.87831518e-01 -2.35077025e-02]
 [-0.00000000e+00 -6.12372436e-01 -3.53553391e-01 -4.87831518e-01
   2.35077025e-02  5.11339220e-01]
 [-0.00000000e+00 -0.00000000e+00 -7.07106781e-01 -2.35077025e-02
  -5.11339220e-01 -4.87831518e-01]
 [ 5.77350269e-01 -4.08248290e-01  2.77555756e-17  6.66390246e-01
  -1.78558728e-01  1.55051026e-01]
 [ 5.77350269e-01  2.04124145e-01 -3.53553391e-01 -1.55051026e-01
   6.66390246e-01 -1.78558728e-01]
 [-0.00000000e+00  6.12372436e-01 -3.53553391e-01  1.78558728e-01
  -1.55051026e-01  6.66390246e-01]]


x=
 [[19.25]
 [10.5 ]
 [16.65]]

res=
 0.09999999999999824
```

Or, we could immediately use the `lstsq` function to solve our system. Note that this function returns the square of the residual.

```
A = np.array([(1.0, 0, 0), (0, 1, 0), (0, 0, 1), (-1, 1, 0), (-1, 0, 1), (0, -
 ↪1, 1)])
b = np.array([[19.3, 10.5, 16.6, -8.7, -2.6, 6.2]]).T

p, res, _, _ = linalg.lstsq(A, b)
print("x=\n", p, "\n\nres=\n", np.sqrt(res))
```

```
x=
 [[19.25]
 [10.5 ]
 [16.65]]

res=
 [0.1]
```

3.4 Rank deficiency

So far we assumed that $\text{rank}(\mathbf{A}) = n$. If this is not the case, you can still perform a QR factorization of $\mathbf{A}$ but the upper triangular matrix will be singular. This means that multiple $\mathbf{x}$ vectors with the same minimal norm exist. Such situations arise when using a wrong model, a corrupt data source, or a poorly designed experiment.

> **Example**
>
> As an example, consider the same experiment of Archimedes, but assume that this time, he only measured the densities of the three objects relative to each other by submerging the scale and recording the differences in apparent weight. It is immediately clear from this information alone, one cannot determine the absolute density of each object. Let's consider the least-squares system:
>
> $$\mathbf{Ax} = \begin{bmatrix} -1 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \cong \begin{bmatrix} -8.7 \\ -2.6 \\ 6.2 \end{bmatrix} = \mathbf{b}$$
>
> It's QR factorization reads:
>
> $$\mathbf{A} = \mathbf{QR} = \begin{bmatrix} -0.70710678 & 0.40824829 & 0.57735027 \\ -0.70710678 & -0.40824829 & -0.57735027 \\ 0 & -0.81649658 & 0.57735027 \end{bmatrix} \begin{bmatrix} 1.41421356 & -0.70710678 & -0.70710678 \\ 0 & 1.22474487 & -1.22474487 \\ 0 & 0 & 0 \end{bmatrix}$$
>
> which indeed shows that $\mathbf{R}$ is singular.
>
> If we want to proceed, we could assign the density of one of the objects to $0\,\text{kg/m}^3$ and find a solution $\mathbf{x} = [2.6, -6.15, 0]$ (the minus sign meaning that the second object (pure silver) is less dense than the crown).
>
> Note that this solution does not exactly satisfy the linear system, as this is impossible since it is inconsistent.

3.5 Singular Value Decomposition

Diagonal linear least square systems are even easier to solve than triangular ones. It is possible to go beyond the triangular QR factorization to achieve a diagonal factorization using a similar approach based on orthogonal transformations. As this is closely related to algorithms for computing eigenvectors, we will not consider this in detail here.

The **singular value decomposition (SVD)** of an $m \times n$ matrix $\mathbf{A}$ has the form

$$\mathbf{A} = \mathbf{U\Sigma V}^\intercal$$

where $\mathbf{U}$ is an $m \times m$ orthogonal matrix, $\mathbf{V}$ is an $n \times n$ orthogonal matrix, and $\mathbf{\Sigma}$ is an $m \times n$ diagonal matrix, with

$$\sigma_{ij} = \begin{cases} 0, & \text{for } i \neq j \\ \sigma_i \geq 0, & \text{for } i = j \end{cases}.$$

The diagonal entries σ_i are called the **singular values** of $\mathbf{A}$ and are usually ordered so that $\sigma_{i-1} \geq \sigma_i, i = 2, \ldots, \min\{m, n\}$, i.e. from largest value (upper left) to smallest value (bottom right). The columns $\mathbf{u}_i$ of $\mathbf{U}$ and $\mathbf{v}_i$ of $\mathbf{V}$ are the corresponding left and right **singular vectors**.

The SVD provides a very flexible method for solving linear least squares problems of any shape or rank. The least-squares solution to $\mathbf{Ax} \cong \mathbf{b}$ of minimum Euclidean norm is given by

$$\mathbf{x} = \sum_{\sigma_i \neq 0} \frac{\mathbf{u}_i^\intercal \mathbf{b}}{\sigma_i} \mathbf{v}_i$$

The SVD is especially useful for ill-conditioned or nearly rank-deficient problems, since any tiny singular values can be dropped from the summation, thereby making the solution much less sensitive to perturbations in the data.

We shall return to the SVD when learning about eigenvalues, because the two concepts are closely related, and postpone a discussion of how to calculate the decomposition matrices until then. For now, we illustrate its practical aspects with some handy applications, including its role in solving linear least squares problems.

Example

The SVD of the matrix $\mathbf{A}$ from the previous example is given by:

$$\mathbf{A} = \begin{bmatrix} -1 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & 1 \end{bmatrix}$$
$$= \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\intercal = \begin{bmatrix} -0.707 & 0.408 & 0.577 \\ -0.707 & -0.408 & -0.577 \\ 0 & -0.816 & 0.577 \end{bmatrix} \begin{bmatrix} 1.732 & 0 & 0 \\ 0 & 1.732 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0.816 & -0.408 & -0.408 \\ 0 & 0.707 & -0.707 \\ 0.577 & 0.577 & -0.577 \end{bmatrix}$$

The least squares solution of minimum norm is therefore given by

$$\mathbf{x} = \frac{\mathbf{u}_1^\intercal \mathbf{b}}{\sigma_1}\mathbf{v}_1 + \frac{\mathbf{u}_2^\intercal \mathbf{b}}{\sigma_2}\mathbf{v}_2 = \frac{7.99}{1.732}\begin{bmatrix} 0.816 \\ -0.408 \\ -0.408 \end{bmatrix} + \frac{-7.55}{1.732}\begin{bmatrix} 0 \\ 0.707 \\ -0.707 \end{bmatrix} = \begin{bmatrix} 3.76 \\ -4.96 \\ 1.19 \end{bmatrix}$$

Note that we only use the nonzero singular values to find $\mathbf{x}$.

```
A = np.array([(-1, 1, 0), (-1, 0, 1), (0, -1, 1)])

U, s, Vh = linalg.svd(A)
with np.printoptions(precision=3, suppress=True):
    print("U=")
    print(U)
    print("\ns=")
    print(s)
    print("\nV=")
    print(Vh)
```

```
U=
[[-0.707  0.408  0.577]
 [-0.707 -0.408 -0.577]
 [-0.    -0.816  0.577]]

s=
[1.732 1.732 0.   ]

V=
[[ 0.816 -0.408 -0.408]
 [-0.     0.707 -0.707]
 [ 0.577  0.577  0.577]]
```

If you compare this with the SVD of the original problem, in which the density of the objects was independently measured, we see that there, we do find 3 non-zero singular values.

```
A = np.array([(1.0, 0, 0), (0, 1, 0), (0, 0, 1), (-1, 1, 0), (-1, 0, 1), (0, -
↪1, 1)])


U, s, Vh = linalg.svd(A)
```

(continues on next page)

(continued from previous page)

```
with np.printoptions(precision=3, suppress=True):
    print("U=")
    print(U)
    print("\ns=")
    print(s)
    print("\nV=")
    print(Vh)
```

```
U=
[[ 0.    -0.408  0.577  0.511  0.488 -0.024]
 [ 0.354  0.204  0.577 -0.488  0.024  0.511]
 [-0.354  0.204  0.577 -0.024 -0.511 -0.488]
 [ 0.354  0.612 -0.     0.666 -0.179  0.155]
 [-0.354  0.612  0.    -0.155  0.666 -0.179]
 [-0.707  0.     0.     0.179 -0.155  0.666]]

s=
[2. 2. 1.]

V=
[[-0.     0.707 -0.707]
 [-0.816  0.408  0.408]
 [ 0.577  0.577  0.577]]
```

3.5.1 Other applications of SVD

The singular value decomposition $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\intercal$ can be used in the following applications as well:

Euclidean matrix norm

As stated before in the linear systems notebook, the matrix norm corresponding to the Euclidean vector norm is equal to the largest singular value of the matrix,

$$\|\mathbf{A}\|_2 = \max_{\mathbf{x}\neq\mathbf{0}} \frac{\|\mathbf{A}\mathbf{x}\|_2}{\|\mathbf{x}\|_2} = \sigma_{\max}$$

Euclidean condition number

The condition number of an arbitrary matrix $\mathbf{A}$, which we saw as a measure to quantify the sensitivity of the solution to changes in the problem data, is given (as opposed to the definition we saw earlier that was only valid for square matrices) by the ratio

$$\text{cond}_2(\mathbf{A}) = \frac{\sigma_{\max}}{\sigma_{\min}}$$

Note that, just as before, we find $\text{cond}_2(\mathbf{A}) = \infty$ for singular matrices, because there, $\sigma_{\min} = 0$.

Rank determination

The rank of a matrix is equal to the number of nonzero singular values it has. In practice, the rank can be not so well-determined because of very small singular values, and then it might be suitable to disregard any values under a certain cut-off number. Although both methods can be used for this purpose, using SVD to determine the rank of a matrix is a more reliable way than using QR factorization. Consider the example below:

Example

Consider the matrix

$$\mathbf{A} = \begin{bmatrix} 0.913 & 0.6559 \\ 0.780 & 0.563 \\ 0.457 & 0.330 \end{bmatrix}$$

When using a QR factorization of $\mathbf{A}$ we find

```
A = np.array([(0.913, 0.659), (0.780, 0.563), (0.457, 0.330)])
_, R = linalg.qr(A)
with np.printoptions(precision=4, suppress=True):
    print(R)
```

```
[[-1.2848 -0.9274]
 [ 0.      0.0001]
 [ 0.      0.    ]]
```

which shows that $\mathbf{R}$ is very close to being nonsingular, and if we use it to solve a least squares problem, the result will be very sensitive to perturbations in the input data. We can also calculate the SVD decomposition of $\mathbf{A}$ to find

```
_, s, _ = linalg.svd(A)
print(np.diag(s))
```

```
[[1.58460342e+00 0.00000000e+00]
 [0.00000000e+00 1.05619406e-04]]
```

Here, we see that one of the singular values is only 0.000106 and the rank would be taken to be 1 rather than 2.

Pseudoinverse

Let's now generalize the definition of a matrix-inverse to matrices that are not necessarily square or non-singular:

- Define the pseudoinverse of a scalar σ as $1/\sigma$ (or 0 if $\sigma = 0$)
- Define the pseudoinverse of a (possibly rectangular) diagonal matrix by transposing the matrix and taking the scalar pseudo-inverse of each entry.

The **pseudoinverse** of a general matrix $\mathbf{A}$ is given by

$$\mathbf{A}^{+} = \mathbf{V}\mathbf{\Sigma}^{+}\mathbf{U}^{\intercal}$$

- If the matrix $\mathbf{A}$ is square and nonsingular this definition agrees with $\mathbf{A}^{-1}$.
- In all cases, the solution to a least squares problem $\mathbf{Ax} \cong \mathbf{b}$ is given by $\mathbf{A}^{+}\mathbf{b}$.

Another (computationally inferior) way to find the pseudo-inverse can be obtained via the normal equations

$$\mathbf{A}^{\intercal}\mathbf{Ax} = \mathbf{A}^{\intercal}\mathbf{b}$$

we see that

$$\mathbf{x} = (\mathbf{A}^{\intercal}\mathbf{A})^{-1}\mathbf{A}^{\intercal}\mathbf{b}$$

is a solution of the least squares problem $\mathbf{Ax} \cong \mathbf{b}$.

Consequently, the pseudoinverse $\mathbf{A}^+$ is also given by

$$\mathbf{A}^+ = (\mathbf{A}^\intercal \mathbf{A})^{-1} \mathbf{A}^\intercal$$

Example

As an example, consider the rank-deficient matrix $\mathbf{A}$ from an earlier example:

$$\mathbf{A} = \begin{bmatrix} -1 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & 1 \end{bmatrix}$$

Using the definition of the pseudoinverse we find

$$\mathbf{A}^+ = \frac{1}{3} \begin{bmatrix} -1 & -1 & 0 \\ 1 & 0 & -1 \\ 0 & 1 & 1 \end{bmatrix}$$

The least squares solution of the problem with $b = [-8.7, -2.6, 6.2]^\intercal$ thus reads:

$$\mathbf{x} = \mathbf{A}^+ \mathbf{b} = \frac{1}{3} \begin{bmatrix} -1 & -1 & 0 \\ 1 & 0 & -1 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} -8.7 \\ -2.6 \\ 6.2 \end{bmatrix} = \begin{bmatrix} 3.76 \\ -4.96 \\ 1.19 \end{bmatrix}$$

The pseudoinverse can be calculated using `scipy` with the `pinv` function:

```
A = np.array([(-1, 1, 0), (-1, 0, 1), (0, -1, 1)])
B = linalg.pinv(A)
print(B)
```

```
[[-3.33333333e-01 -3.33333333e-01 -3.40644025e-18]
 [ 3.33333333e-01  7.75896271e-17 -3.33333333e-01]
 [-6.29681990e-17  3.33333333e-01  3.33333333e-01]]
```

The resulting solution to the least squares problem thus is

```
b = np.array([(-8.7, -2.6, 6.2)]).T
print(B @ b)
```

```
[[ 3.76666667]
 [-4.96666667]
 [ 1.2       ]]
```

In agreement with the solution obtained directly with `linalg.lstsq`:

```
A = np.array([(-1, 1, 0), (-1, 0, 1), (0, -1, 1)])
b = np.array([(-8.7, -2.6, 6.2)]).T
x, _, _, _ = linalg.lstsq(A, b)
print(x)
```

```
[[ 3.76666667]
 [-4.96666667]
 [ 1.2       ]]
```

3.6 Sensitivity and condition number

Similar to the case of systems of linear equations with an exact solution, we want to find some measure to quantify the sensitivity of the solution of least squares problems to changes in the problem data.

Generalizing the definition of a condition number to an $m \times n$ matrix with $\text{rank}(\mathbf{a}) = n$, we define

$$\text{cond}(\mathbf{A}) = \|\mathbf{A}\|_2 \cdot \|\mathbf{A}^+\|_2$$

By convention, $\text{cond}(\mathbf{A}) = \infty$ if $\text{rank}(\mathbf{A}) < n$

Let's now also generalize the expression,

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}'\|} \leq \text{cond}(\mathbf{A}) \frac{\|\mathbf{r}\|}{\|\mathbf{A}\| \cdot \|\mathbf{x}'\|}$$

which we found for square systems.

We start from the solution $\mathbf{x}$ to the least squares problem, i.e.

$$\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2 \text{ is minimal}$$

Similarly, denote with $\mathbf{x}' = \mathbf{x} + \mathbf{\Delta x}$ the solution to a problem with a perturbed right hand side $\mathbf{b} + \mathbf{\Delta b}$:

$$\|\mathbf{A}(\mathbf{x} + \mathbf{\Delta x}) - (\mathbf{b} + \mathbf{\Delta b})\|_2 \text{ is minimal}$$

From the normal equations we know that

$$\mathbf{A}^\intercal\mathbf{A}(\mathbf{x} + \mathbf{\Delta x}) = \mathbf{A}^\intercal(\mathbf{b} + \mathbf{\Delta b})$$

Thus,

$$\mathbf{\Delta x} = \mathbf{A}^+\mathbf{\Delta b}$$

Taking norms and dividing by $\|\mathbf{x}\|_2$ we find

$$\frac{\|\mathbf{\Delta x}\|_2}{\|\mathbf{x}\|_2} \leq \frac{\|\mathbf{A}^+\|_2\|\mathbf{\Delta b}\|_2}{\|\mathbf{x}\|_2}$$

Using the definition $\text{cond}(\mathbf{A}) = \|\mathbf{A}\|_2 \cdot \|\mathbf{A}^+\|_2$ and multiplying both the denominator and enumerator in the right-hand-side with $\|\mathbf{b}\|_2$ we find

$$\frac{\|\mathbf{\Delta x}\|_2}{\|\mathbf{x}\|_2} \leq \frac{\text{cond}(\mathbf{A})}{\|\mathbf{A}\|_2} \frac{\|\mathbf{b}\|_2\|\mathbf{\Delta b}\|_2}{\|\mathbf{b}\|_2\|\mathbf{x}\|_2} \leq \text{cond}(\mathbf{A}) \frac{\|\mathbf{\Delta b}\|_2}{\|\mathbf{b}\|_2} \frac{\|\mathbf{b}\|_2}{\|\mathbf{A}\mathbf{x}\|_2} = \text{cond}(\mathbf{A}) \frac{1}{\cos(\theta)} \frac{\|\mathbf{\Delta b}\|_2}{\|\mathbf{b}\|_2}$$

where $\cos(\theta) = \frac{\|\mathbf{A}\mathbf{x}\|_2}{\|\mathbf{b}\|_2}$ denotes the cosine of the angle θ between the vectors $\mathbf{b}$ and $\mathbf{A}\mathbf{x}$.

> To see why this is true, start from the inner product
>
> $$(\mathbf{A}\mathbf{x})^\intercal\mathbf{b} = \|\mathbf{A}\mathbf{x}\|_2\|\mathbf{b}\|_2 \cos(\theta)$$
>
> rearrange to isolate $\cos(\theta)$
>
> $$\cos(\theta) = \frac{(\mathbf{A}\mathbf{x})^\intercal\mathbf{b}}{\|\mathbf{A}\mathbf{x}\|_2\|\mathbf{b}\|_2}$$
>
> make use of the normal equation to find
>
> $$\cos(\theta) = \frac{\mathbf{x}^\intercal\mathbf{A}^\intercal\mathbf{b}}{\|\mathbf{A}\mathbf{x}\|_2\|\mathbf{b}\|_2} = \frac{\mathbf{x}^\intercal\mathbf{A}^\intercal\mathbf{A}\mathbf{x}}{\|\mathbf{A}\mathbf{x}\|_2\|\mathbf{b}\|_2} = \frac{(\mathbf{A}\mathbf{x})^\intercal(\mathbf{A}\mathbf{x})}{\|\mathbf{A}\mathbf{x}\|_2\|\mathbf{b}\|_2}$$

Recognize that the numerator represents the square of the Euclidean norm of the vector $\mathbf{Ax}$

$$\cos(\theta) = \frac{\|\mathbf{Ax}\|_2^2}{\|\mathbf{Ax}\|_2\|\mathbf{b}\|_2} = \frac{\|\mathbf{Ax}\|_2}{|\mathbf{b}\|_2}$$

In contrast to the result for square matrices, it does no longer hold that the sensitivity of the solution only depends on the matrix $\mathbf{A}$, but now also depends on the right-hand-side vector $\mathbf{b}$.

Example

Let's verify the expression above by revisiting the experiment of Archimedes where we encountered the following linear least squares problem:

$$\mathbf{Ax} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ -1 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} \rho_G \\ \rho_S \\ \rho_C \end{bmatrix} \cong \begin{bmatrix} 19.3 \\ 10.5 \\ 16.6 \\ -8.7 \\ -2.6 \\ 6.2 \end{bmatrix} = \mathbf{b}$$

The solution to this problem was $\mathbf{x} = [19.25 \quad 10.50 \quad 16.65]^\intercal$. If a perturbation $\mathbf{\Delta b}$ is applied we get the following problem:

$$\mathbf{Ax}' = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ -1 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} \rho'_G \\ \rho'_S \\ \rho'_C \end{bmatrix} \cong \begin{bmatrix} 19.3 \\ 10.5 \\ 16.6 \\ -8.7 \\ -2.6 \\ 6.2 \end{bmatrix} + \begin{bmatrix} 0.5 \\ -1 \\ 1 \\ -0.5 \\ 2 \\ -1 \end{bmatrix} = \mathbf{b} + \mathbf{\Delta b}$$

The solution of this perturbed linear least squares problem is $\mathbf{x}' = [19.125 \quad 10.50 \quad 17.275]^\intercal$. With this information, the Eucledian norms of $\mathbf{x}$, $\mathbf{\Delta x}$, $\mathbf{b}$, $\mathbf{\Delta b}$ and $\mathbf{Ax}$ can be calculated resulting in:

$$\frac{\|\mathbf{\Delta x}\|_2}{\|\mathbf{x}\|_2} = 0.023, \qquad \frac{\|\mathbf{\Delta b}\|_2}{\|\mathbf{b}\|_2} = 0.092, \qquad \cos\theta = \frac{\|\mathbf{Ax}\|_2}{\|\mathbf{b}\|_2} \approx 1$$

Finding that $\cos\theta \approx 1$ is a good sign that your experiment was well set-up and resulted in measurements that are quite consistent with each other.

The condition number can be obtained by finding the pseudoinverse of $\mathbf{A}$ and using the following equation:

$$\text{cond}_2(\mathbf{A}) = \|\mathbf{A}\|_2 \cdot \|\mathbf{A}^+\|_2, = 2 \times 1 = 2$$

Or, alternatively, by performing the singular value decomposition of the matrix $\mathbf{A}$ and using:

$$\text{cond}_2(\mathbf{A}) = \frac{\sigma_{\max}}{\sigma_{\min}} = \frac{2}{1} = 2$$

Finally, this results in:

$$0.023 = \frac{\|\mathbf{\Delta x}\|_2}{\|\mathbf{x}\|_2} \leq \text{cond}(\mathbf{A}) \frac{1}{\cos(\theta)} \frac{\|\mathbf{\Delta b}\|_2}{\|\mathbf{b}\|_2} = 0.184$$

3.7 Which method to use?

We've seen three different methods (solving the normal equations, using householder transformations, or using SVD) with which you can solve a least-squares problem. Which one is your best choice depends on a trade-off between efficiency, accuracy and reliability.

- The easiest method to implement are the normal equations (which only require matrix multiplications and Cholesky decomposition). However, this method is computationally quite expensive and the error is proportional to $[\text{cond}(\mathbf{A})]^2$, which means it can break down quite easily
- The most efficient and accurate orthogonalization method (for dense matrices at least) is typically the Householder method. For square systems, it requires about the same amount of work as the normal equations, but for strongly overdetermined systems, it becomes only about half as efficient. On the other hand, it is much more broadly applicable due to its better accuracy.
- SVD is the most expensive method, but also offers superb robustness and reliability.

The `linalg.lstsq` function allows the user to choose the LAPACK driver :

> lapack_driverstr, optional
>
> Which LAPACK driver is used to solve the least-squares problem. Options are 'gelsd', 'gelsy', 'gelss'. Default >('gelsd') is a good choice. However, 'gelsy' can be slightly faster on many problems. 'gelss' was used > historically. It is generally slow but uses less memory.

Until 2015, the default choice was gelss which uses an SVD algorithm. Since then, the default choice (gelsd) uses a very efficient "divide and conquer" implementation of SVD, which is beyond the scope of this course. The other option, gelsy, is based on a QR factorization which is faster than the SVD-based choices, but is not chosen as a default because of its lower reliability.

3.8 Further information

More information on solving least squares systems with `scipy` can be found on

https://docs.scipy.org/doc/scipy/tutorial/linalg.html#solving-linear-least-squares-problems-and-pseudo-inverses

CHAPTER 4

Eigenvalue Problems

```
import matplotlib.pyplot as plt
import numpy as np
from scipy import integrate, linalg
```

4.1 Introduction, concept and useful properties

A given direction in a vector space is determined by any nonzero vector pointing in that direction. Given an $n \times n$ matrix $\mathbf{A}$ representing a linear transformation on an n-dimensional vector space, we wish to find a nonzero vector $\mathbf{x}$ and a scalar λ such that

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

Such a scalar λ is called an **eigenvalue**, and $\mathbf{x}$ is a corresponding **eigenvector**.

An eigenvector of a matrix determines a direction in which the effect of the matrix is particularly simple: The matrix expands or shrinks any vector lying in that direction by a scalar multiple, and the expansion or contraction factor is given by the corresponding eigenvalue λ. Thus, eigenvalues and eigenvectors provide a means of understanding the complicated behavior of a general linear transformation by decomposing it into simpler actions.

4.1.1 Characteristic polynomial

For a square matrix $\mathbf{A}$, the equation $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$ is equivalent to

$$(\mathbf{A} - \lambda\mathbf{I})\mathbf{x} = \mathbf{0}$$

The eigenvalues of $\mathbf{A}$ are the values of λ such that

$$\det(\mathbf{A} - \lambda\mathbf{I}) = 0$$

The polynomial $p(\lambda) = \det(\mathbf{A} - \lambda\mathbf{I})$ is called the **characteristic polynomial** of $\mathbf{A}$ and its roots are the eigenvalues of $\mathbf{A}$.

Example

Consider the matrix

$$\begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix}$$

The characteristic polynomial is

$$\det\left(\begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix} - \lambda \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right) = \det\left(\begin{bmatrix} 3-\lambda & 1 \\ 1 & 3-\lambda \end{bmatrix}\right)$$

$$= (3-\lambda)(3-\lambda) - (1)(1) = \lambda^2 - 6\lambda + 8 = 0$$

The roots of this polynomial (and hence the eigenvalues of $\mathbf{A}$) are 4 and 2.

```
# Compute eigenvalues with this technique
A = np.array([[3, 1], [1, 3]])


def char_pol(A):
    """compute the characteristic polynomial of a matrix

    Parameters
    ----------
    A: a square matrix

    Returns
    -------
    A string that shows the characteristic polynomial with X the eigenvalues.
    """

    # First we will check the dimensions of the array
    if np.shape(A)[0] != np.shape(A)[1]:
        raise TypeError("Matrix A must be a square matrix.")

    # Now we will use the .poly() function imported from numpy
    # to find the characteristic polynomial.
    return np.poly(A)


A = char_pol(A)


def format_poly(A):
    """Return the characteristic polynomial of matrix A formatted as a string.
  ↪"""
    string = ""
    n = len(A) - 1
    for el in A:
        string += f"{el}X^{n} + "
        n -= 1
    return string.rstrip("+ ")
```

(continues on next page)

(continued from previous page)

```
poly = format_poly(A)

# The next function will compute the roots of the characteristic polynomial␣
 ↪of A
# (i.e. the eigenvalues of A).
eigenvalues = np.roots(A)

print(f"The characteristic polynomial of matrix A is given by {poly}, ")
print(f"and the corresponding eigenvalues are {eigenvalues}.")
```

```
The characteristic polynomial of matrix A is given by 1.0X^2 + -6.0X^1 + 8.0X^
 ↪0,
and the corresponding eigenvalues are [4. 2.].
```

Although, in theory, this is a nice way to find the eigenvalues of a matrix $\mathbf{A}$, calculating the roots of its characteristic polynomial is not a good numerical way to find the eigenvalues of a matrix of nontrivial size for several reasons

- Computing the coefficients of the characteristic polynomial for a large matrix is in itself already a substantial task
- The coefficients of the characteristic polynomial can be highly sensitive to small perturbations in $\mathbf{A}$ which can render their computation instable
- Rounding errors in finding the characteristic polynomial can destroy the accuracy of the roots
- Computing the roots of a polynomial of high degree is a nontrivial and substantial task

4.1.2 Properties and transformations

Many numerical methods for computing eigenvalues and eigenvectors are based on reducing the original matrix to a simpler form, whose eigenvalues and eigenvectors are then easily determined. Thus, we need to identify what types of transformations preserve eigenvalues, and for what types of matrices the eigenvalues are easily determined.

- If $\mathbf{A}$ is symmetric/Hermitian, all its eigenvalues are real.
- **Shift**: if $\mathbf{Ax} = \lambda\mathbf{x}$ and σ any scalar, then $(\mathbf{A} - \sigma\mathbf{I})\mathbf{x} = (\lambda - \sigma)\mathbf{x}$; The eigenvalues are shifted by σ, but the eigenvectors remain unchanged.
- **Inversion**: $\mathbf{A}^{-1}$ has the same eigenvectors as $\mathbf{A}$, and eigenvalues $1/\lambda$
- **Powers**: $\mathbf{A}^k$ has the same eigenvectors as $\mathbf{A}$, and eigenvalues λ^k
- **Polynomials**: for a general polynomial $p(t)$, $p(\mathbf{A})\mathbf{x} = p(\lambda)\mathbf{x}$. Thus the eigenvalues of a polynomial in a matrix $\mathbf{A}$ are given by the same polynomial, evaluated at the eigenvalues of $\mathbf{A}$ and the corresponding eigenvectors remain the same as those of $\mathbf{A}$.
- **Similarity**: A matrix $\mathbf{B}$ is *similar* to a matrix $\mathbf{A}$ if there exists an invertible matrix $\mathbf{T}$ such that
-

$$\mathbf{B} = \mathbf{T}^{-1}\mathbf{AT}$$

It follows that:

$$\mathbf{By} = \lambda\mathbf{y} \Rightarrow \mathbf{T}^{-1}\mathbf{ATy} = \lambda\mathbf{y} \Rightarrow \mathbf{ATy} = \lambda\mathbf{Ty}$$

In other words, $\mathbf{B} = \mathbf{T}^{-1}\mathbf{AT}$ has the same eigenvalues as $\mathbf{A}$, but systematically transforms its eigenvectors.

4.2 Calculating eigenvalues and eigenvectors

4.2.1 Power iteration

This is a simple, but limited method that allows to estimate the dominant eigenvalue and its corresponding eigenvector.

It works by multiplying an arbitrary nonzero vector repeatedly by the matrix.

Assuming that $\mathbf{A}$ has a unique eigenvalue λ_1 of maximum modulus, with corresponding eigenvector $\mathbf{v}_1$, power iteration converges to a multiple of $\mathbf{v}_1$.

> **Proof:**
>
> Assume that we can express the starting vector $\mathbf{x}_0$ as a linear combination $\mathbf{x}_0 = \sum_{j=1}^{n} \alpha_j \mathbf{v}_j$, with $\mathbf{v}_j$ the eigenvectors of $\mathbf{A}$.
>
> $$\begin{aligned}\mathbf{x}_k &= \mathbf{A}\mathbf{x}_{k-1} = \mathbf{A}^2\mathbf{x}_{k-2} = \ldots = \mathbf{A}^k\mathbf{x}_0 \\ &= \mathbf{A}^k \sum_{j=1}^{n} \alpha_j \mathbf{v}_j = \sum_{j=1}^{n} \alpha_j \mathbf{A}^k \mathbf{v}_j = \sum_{j=1}^{n} \lambda_j^k \alpha_j \mathbf{v}_j \\ &= \lambda_1^k \left(\alpha_1 \mathbf{v}_1 + \sum_{j=2}^{n} (\lambda_j/\lambda_1)^k \alpha_j \mathbf{v}_j \right)\end{aligned}$$
>
> Here is $|\lambda_j/\lambda_1| < 1$ since λ_1 is of maximum modulus. As a result, this factor will converge to 0 when k becomes large.

Power iteration usually works well in practice, but might fail for the following reasons:

- The starting vector $\mathbf{x}_0$ may have *no* component in the dominant eigenvector $\mathbf{v}_1$. In practice this is very unlikely and is mitigated after a few iterations due to rounding errors that introduce such a component.
- There may be more than 1 eigenvalue with the same maximum modulus, in which case the algorithm might converge to a linear combination of the corresponding eigenvectors.
- For a real matrix and real starting vector, the iteration can never converge to a complex vector.

Geometric growth of the components at each iteration risks overflow or underflow, so in practice the approximate eigenvector is rescaled to have norm 1 at every iteration. Then, $\mathbf{x}_k \to \mathbf{v}_1/\|\mathbf{v}_1\|_\infty$ and $\|\mathbf{y}_k\|_\infty \to \|\lambda_1\|$. With $\mathbf{A}\mathbf{x}_k = \mathbf{y}_k$ and using the infinity norm defined as $\|\mathbf{a}\|_\infty = max(|a_1|, |a_2|, \ldots, |a_n|)$.

The convergence rate of power iteration is linear (and proportional with $\|\lambda_2/\lambda_1\|$, where λ_2 is the eigenvalue with second largest modulus.

A straightforward implementation is shown below:

```
def power(A):
    """Normalized power iteration."""
    # x = np.random.random(len(A))
    x = np.array([0, 1])
    vectors = [x]
    for _ in range(15):
        y = A @ x
        x = y / linalg.norm(y, np.inf)
        print(f"[{x[0]:5.3f}, {x[1]:5.3f}]: {linalg.norm(y, np.inf):.3f}")
        vectors.append(x)
    return x, vectors
```

(continues on next page)

(continued from previous page)

```
# Example
A = np.array([[1, 3], [3, 1]])
iteration = power(A)[1]
# the actual eigenvectors v_0 and v_1, solved by hand
eigenvectors = np.array([1, 1]), np.array([1, -1])
```

```
[1.000, 0.333]: 3.000
[0.600, 1.000]: 3.333
[1.000, 0.778]: 3.600
[0.882, 1.000]: 3.778
[1.000, 0.939]: 3.882
[0.969, 1.000]: 3.939
[1.000, 0.984]: 3.969
[0.992, 1.000]: 3.984
[1.000, 0.996]: 3.992
[0.998, 1.000]: 3.996
[1.000, 0.999]: 3.998
[1.000, 1.000]: 3.999
[1.000, 1.000]: 4.000
[1.000, 1.000]: 4.000
[1.000, 1.000]: 4.000
```

As expected, this converges to the eigenvalue 4, and the corresponding eigenvector `[1, 1]`.

4.2.2 Inverse iteration

For some applications, we're interested in the smallest eigenvalue of a matrix. Then we can make use of the fact that the eigenvalues of $\mathbf{A}^{-1}$ are $1/\lambda$. This suggests to use power iteration on the inverse of $\mathbf{A}$, but as usual the inverse of $\mathbf{A}$ does not need to be calculated explicitly.

Instead, the equivalent system of linear equations is solved at each iteration using the triangular factors resulting from e.g. LU-factorization of $\mathbf{A}$, which need only to be calculated once. Using $\mathbf{L}$ and $\mathbf{U}$, we can then efficiently solve $\mathbf{Ay} = \mathbf{x}$ using forward and backward substitution. (These functions are also used for the Rayleigh quotient iteration below.)

```
# define helper functions for the forward and backward substitution
def forward_substitution(L, b):
    n = len(L)
    x = np.zeros(n)
    for j in range(n):
        if L[j][j] == 0:  # stop if matrix is singular
            break
        x[j] = b[j] / L[j][j]
        for i in range(j, n):
            b[i] = b[i] - L[i][j] * x[j]
    return x


def backward_substitution(U, b):
    n = len(U)
    x = np.zeros(n)
    # Notice that the last value of range is exclusive,
    # which is very counter-intuitive for countdowns).
    for j in range(n - 1, -1, -1):
        if U[j][j] == 0:  # stop if matrix is singular
```

(continues on next page)

(continued from previous page)

```
            break
        x[j] = b[j] / U[j][j]
        for i in range(0, j):
            b[i] = b[i] - U[i][j] * x[j]
    return x
```

Inverse iteration converges to the eigenvector corresponding to the largest eigenvalue of $\mathbf{A}^{-1}$, which is the smallest eigenvalue of $\mathbf{A}$.

```
def inverse_iter(A):
    num_iters = 50
    tol = 1e-10

    _, L, U = linalg.lu(A, permute_l=False)

    # Initialize a random starting vector x and normalize it
    x = np.random.random(len(A))
    x /= linalg.norm(x, np.inf)  # Normalize x to avoid scaling issues

    # Lists to store the sequence of approximate eigenvectors and eigenvalues
    eigvecs = [x.copy()]
    eigvals = [0]

    for _ in range(num_iters):
        # Solve the system A * y = x using LU decomposition, where A = L*U.
        # This involves forward and backward substitution.
        y = forward_substitution(L, x)  # Solves L * y = x for y
        y = backward_substitution(U, y)  # Solves U * y = y for y

        # Normalize the resulting vector to avoid numerical overflow or␣
↪underflow
        x = y / linalg.norm(y, np.inf)

        # Append the current eigenvector and eigenvalue approximation
        eigvecs.append(x.copy())
        eigvals.append(linalg.norm(y, np.inf))

        # Check for convergence
        if np.abs(eigvals[-1] - eigvals[-2]) < tol:
            print("converged after", len(eigvals), "iterations")
            break

    return np.array(eigvecs), eigvals
```

```
A = np.array([[3, 1], [1, 3]])
inverse_iter(A)
```

```
converged after 38 iterations
```

```
(array([[ 0.90714153,  1.        ],
        [ 0.82252318,  1.        ],
        [ 0.67397711,  1.        ],
        [ 0.43934706,  1.        ],
```

(continues on next page)

(continued from previous page)

```
       [ 0.12420316,  1.        ],
       [-0.21816232,  1.        ],
       [-0.51410924,  1.        ],
       [-0.72346291,  1.        ],
       [-0.85146242,  1.        ],
       [-0.92286692,  1.        ],
       [-0.96067515,  1.        ],
       [-0.98014235,  1.        ],
       [-0.99002164,  1.        ],
       [-0.99499834,  1.        ],
       [-0.99749604,  1.        ],
       [-0.99874724,  1.        ],
       [-0.99937342,  1.        ],
       [-0.99968666,  1.        ],
       [-0.99984332,  1.        ],
       [-0.99992166,  1.        ],
       [-0.99996083,  1.        ],
       [-0.99998041,  1.        ],
       [-0.99999021,  1.        ],
       [-0.9999951 ,  1.        ],
       [-0.99999755,  1.        ],
       [-0.99999878,  1.        ],
       [-0.99999939,  1.        ],
       [-0.99999969,  1.        ],
       [-0.99999985,  1.        ],
       [-0.99999992,  1.        ],
       [-0.99999996,  1.        ],
       [-0.99999998,  1.        ],
       [-0.99999999,  1.        ],
       [-1.        ,  1.        ],
       [-1.        ,  1.        ],
       [-1.        ,  1.        ],
       [-1.        ,  1.        ],
       [-1.        ,  1.        ]]),
[0,
 np.float64(0.26160730851478836),
 np.float64(0.2721846029086229),
 np.float64(0.2907528616085434),
 np.float64(0.32008161739678975),
 np.float64(0.35947460520657293),
 np.float64(0.402270290614358),
 np.float64(0.4392636545216982),
 np.float64(0.46543286426438135),
 np.float64(0.48143280245677744),
 np.float64(0.49035836494289897),
 np.float64(0.4950843935036044),
 np.float64(0.4975177936525079),
 np.float64(0.49875270473782),
 np.float64(0.4993747927327854),
 np.float64(0.49968700499288665),
 np.float64(0.4998434044692047),
 np.float64(0.4999216777047596),
 np.float64(0.4999608327170368),
 np.float64(0.49998041482432215),
```

(continues on next page)

(continued from previous page)

```
  np.float64(0.49999020702856695),
  np.float64(0.4999951034183793),
  np.float64(0.4999975516852129),
  np.float64(0.4999987758366122),
  np.float64(0.49999938791680754),
  np.float64(0.4999996939580291),
  np.float64(0.4999998469789209),
  np.float64(0.499999923489437),
  np.float64(0.4999999617447126),
  np.float64(0.4999999808723549),
  np.float64(0.4999999904361771),
  np.float64(0.49999999521808847),
  np.float64(0.49999999760904423),
  np.float64(0.4999999988045221),
  np.float64(0.49999999940226103),
  np.float64(0.49999999970113046),
  np.float64(0.49999999985056526),
  np.float64(0.49999999992528266)])
```

As expected this converges to [-1, 1] which is the eigenvector corresponding to
the dominant eigenvalue of A^{-1} is 0.5. This corroborates, what we already knew, i.e. the smallest eigenvalue of A is 2.

By shifting the matrix $\mathbf{A}$ to $\mathbf{A} - \sigma\mathbf{I}$, all eigenvalues are also shifted by σ. In case of inverse iteration this approach gives some flexibility in which eigenvalue will be found. If we apply inverse iteration on the matrix $\mathbf{A} - \sigma\mathbf{I}$, the largest eigenvalue of its inverse will be found. The inverse of the eigenvalue gives than the smallest eigenvalue of $\mathbf{A} - \sigma\mathbf{I}$. If we now add σ to the eigenvalue, we find the eigenvalue of $\mathbf{A}$ closest to σ. Also, when the shift is already a close approximation of the eigenvalue, the convergence is very rapid.

```
A = np.array([[3, 1], [1, 3]]) - np.array([[3.8, 0], [0, 3.8]])
inverse_iter(A)
```

```
converged after 13 iterations
```

```
(array([[1.        , 0.85786695],
        [1.        , 0.98314255],
        [1.        , 0.99811281],
        [1.        , 0.99979014],
        [1.        , 0.99997668],
        [1.        , 0.99999741],
        [1.        , 0.99999971],
        [1.        , 0.99999997],
        [1.        , 1.        ],
        [1.        , 1.        ],
        [1.        , 1.        ],
        [1.        , 1.        ],
        [1.        , 1.        ]]),
 [0,
  np.float64(4.684148771412239),
  np.float64(4.962538998367872),
  np.float64(4.99580624604053),
  np.float64(4.999533636174805),
  np.float64(4.999948176963519),
  np.float64(4.999994241825149),
  np.float64(4.999999360202054),
```

(continues on next page)

(continued from previous page)

```
  np.float64(4.9999999289113255),
  np.float64(4.999999992101254),
  np.float64(4.999999999122357),
  np.float64(4.99999999990248),
  np.float64(4.999999999989161)])
```

When using shifted inverse iteration, the value obtained is the **inverse of the shifted eigenvalue**. To find the corresponding eigenvalue of **A**, first take the reciprocal of the result, then add the shift σ back. This final value is the eigenvalue of **A** closest to σ.

In the case of the example above, the eigenvalue is 1/5 + 3.8=4, as expected.

4.2.3 Rayleigh quotient iteration

Given an approximate eigenvector **x** for a real matrix **A**, finding the best estimate for the corresponding eigenvalue λ can be considered as a linear least squares approximation problem:

$$\mathbf{x}\lambda \cong \mathbf{A}\mathbf{x}$$

It's solution, the **Rayleigh quotient**, is given by

$$\lambda = \frac{\mathbf{x}^\mathsf{T}\mathbf{A}\mathbf{x}}{\mathbf{x}^\mathsf{T}\mathbf{x}}$$

This is a better approximation for the eigenvalue than the one obtained at each stage in the power iteration algorithm.

Given an approximate eigenvector, the Rayleigh quotient provides a good estimate for the corresponding eigenvalue. Conversely, inverse iteration converges very rapidly to an eigenvector if an approximate eigenvalue is used as shift. When combining these ideas, we arrive at **Rayleigh quotient iteration**.

An example implementation is shown below.

```
def rayleigh_iter(A):
    # x = np.random.random(len(A))
    x = np.array([2, 0.05])
    eigvecs = [x.copy()]
    eigvals = [0]
    while True:
        shift = (x @ A @ x) / (x @ x)
        B = A - shift * np.identity(len(A))
        _, L, U = linalg.lu(B, permute_l=False)
        y = forward_substitution(L, x)
        y = backward_substitution(U, y)
        if linalg.norm(y, np.inf) != 0:
            x = y / linalg.norm(y, np.inf)
            eigvecs.append(x.copy())
            eigvals.append(shift)
        else:
            # the iteration will halt as the shifted matrix
            # becomes singular (eigenvalue = 0)
            break
    return np.array(eigvecs), eigvals


A = np.array([[3.0, 1.0], [1.0, 3.0]])
rayleigh_iter(A)
```

```
(array([[2.        , 0.05      ],
        [1.        , 0.07487523],
        [1.        , 0.22132306],
        [1.        , 0.58835125],
        [1.        , 0.96578033],
        [1.        , 0.99998945],
        [1.        , 1.        ]]),
 [0,
  np.float64(3.04996876951905),
  np.float64(3.148915602234091),
  np.float64(3.421976093726194),
  np.float64(3.8741196801386306),
  np.float64(3.9993941292835142),
  np.float64(3.9999999999443485)])
```

Note that we quickly converge to the eigenvector [1, 1] with eigenvalue 4(quicker than with power iteration).

4.2.4 Deflation

The process of **deflation** removes a known eigenvalue from a matrix, so that further eigenvalues and eigenvectors can be determined. This process is similar to removing a known root λ_1 from a polynomial $p(\lambda)$ by dividing it out to obtain $p(\lambda)/(\lambda - \lambda_1)$.

This can be achieved by letting $\mathbf{u}_1$ be any vector such that $\mathbf{u}_1^{\mathrm{T}}\mathbf{x}_1 = \lambda_1$. Then the matrix $\mathbf{A} - \mathbf{x}_1\mathbf{u}_1^{\mathrm{T}}$ has eigenvalues $0, \lambda_2, \ldots, \lambda_n$.

> An **example** of a deflation procedure is shown below to find both eigenvalues of the matrix
>
> $$\begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix}$$

Note that the eigenvector found depends on the choice of $\mathbf{u}_1$, when using this procedure and **the remaining eigenvectors of the deflated matrix are generally different from those of the original matrix**. This is why deflation is often limited to theoretical applications, and practical computations of multiple eigenvalues are usually performed with other methods, such as shifted inverse iteration.

We're not going to look deeper into this procedure because

- it becomes increasingly cumbersome and numerically less accurate to find eigenvalues using deflation (so that inverse iteration using the estimated eigenvalues as a shift are necessary)
- there are better ways to find many eigenvalues of a matrix.

```
# We start by finding the largest eigenvalue of A using power iteration
# which gives us "4"
A = np.array([[3, 1], [1, 3]])
print("Original Matrix A:")
print(A)
print("\nApplying Power Iteration to A:")
power(A);
```

```
Original Matrix A:
[[3 1]
 [1 3]]
```

(continues on next page)

(continued from previous page)

```
Applying Power Iteration to A:
[0.333, 1.000]: 3.000
[0.600, 1.000]: 3.333
[0.778, 1.000]: 3.600
[0.882, 1.000]: 3.778
[0.939, 1.000]: 3.882
[0.969, 1.000]: 3.939
[0.984, 1.000]: 3.969
[0.992, 1.000]: 3.984
[0.996, 1.000]: 3.992
[0.998, 1.000]: 3.996
[0.999, 1.000]: 3.998
[1.000, 1.000]: 3.999
[1.000, 1.000]: 4.000
[1.000, 1.000]: 4.000
[1.000, 1.000]: 4.000
```

We now perform deflation to remove the largest eigenvalue $\lambda_1 = 4$ using two different choices of $\mathbf{u}_1$.

First Choice of $\mathbf{u}_1$

We choose:

$$\mathbf{u}_1 = \begin{bmatrix} 0 \\ 4 \end{bmatrix}, \quad \mathbf{x}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

```
u = np.array([[0], [4]])
x = np.array([[1], [1]])
print("\nFirst Choice for u:")
print("u =", u.flatten())
print("\nVerifying u^\intercal x = lambda_1:")
print("u^\intercal x =", np.dot(u.T, x))

# Perform deflation
print("\nDeflating A with x * u^\intercal:")
A_deflated = A - np.dot(x, u.T)
print(A_deflated)
print("\nApplying Power Iteration to Deflated A:")
power(A_deflated);
```

```
First Choice for u:
u = [0 4]

Verifying u^\intercal x = lambda_1:
u^\intercal x = [[4]]

Deflating A with x * u^\intercal:
[[ 3 -3]
 [ 1 -1]]

Applying Power Iteration to Deflated A:
[-1.000, -0.333]: 3.000
[-1.000, -0.333]: 2.000
[-1.000, -0.333]: 2.000
```

(continues on next page)

(continued from previous page)

```
[-1.000, -0.333]: 2.000
[-1.000, -0.333]: 2.000
[-1.000, -0.333]: 2.000
[-1.000, -0.333]: 2.000
[-1.000, -0.333]: 2.000
[-1.000, -0.333]: 2.000
[-1.000, -0.333]: 2.000
[-1.000, -0.333]: 2.000
[-1.000, -0.333]: 2.000
[-1.000, -0.333]: 2.000
[-1.000, -0.333]: 2.000
[-1.000, -0.333]: 2.000
```

Second Choice of $\mathbf{u}_1$

Now we choose a different vector $\mathbf{u}_1$:

$$\mathbf{u}_1 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$$

Repeating the steps with this choice of $\mathbf{u}_1$:

```
A = np.array([[3, 1], [1, 3]])
print("\nSecond Choice for u:")
u = np.array([[2], [2]])
x = np.array([[1], [1]])
print("u =", u.flatten())
print("\nVerifying u^\intercal x = lambda_1:")
print("u^\intercal x =", np.dot(u.T, x))

# Perform deflation
print("\nDeflating A with x * u^\intercal:")
A_deflated = A - np.dot(x, u.T)
print(A_deflated)
print("\nApplying Power Iteration to Deflated A:")
power(A_deflated);
```

```
Second Choice for u:
u = [2 2]

Verifying u^\intercal x = lambda_1:
u^\intercal x = [[4]]

Deflating A with x * u^\intercal:
[[ 1 -1]
 [-1  1]]

Applying Power Iteration to Deflated A:
[-1.000, 1.000]: 1.000
[-1.000, 1.000]: 2.000
[-1.000, 1.000]: 2.000
[-1.000, 1.000]: 2.000
[-1.000, 1.000]: 2.000
[-1.000, 1.000]: 2.000
[-1.000, 1.000]: 2.000
```

(continues on next page)

(continued from previous page)

```
[-1.000, 1.000]: 2.000
[-1.000, 1.000]: 2.000
[-1.000, 1.000]: 2.000
[-1.000, 1.000]: 2.000
[-1.000, 1.000]: 2.000
[-1.000, 1.000]: 2.000
[-1.000, 1.000]: 2.000
[-1.000, 1.000]: 2.000
```

4.2.5 QR Iteration

In practice, the fastest and most used method to find the eigenvalues of a matrix is **QR iteration**. Starting from a matrix $\mathbf{A}$, we define the following sequence:

$$\mathbf{A}_m = \mathbf{Q}_m \mathbf{R}_m$$
$$\mathbf{A}_{m+1} = \mathbf{R}_m \mathbf{Q}_m$$

With $\mathbf{Q}$ an orthogonal matrix and an $\mathbf{R}$ an upper-triangular matrix. This sequence will converge to a triangular matrix with the eigenvalues of $\mathbf{A}$ on its diagonal, or a near-triangular form, which easily allows calculating the eigenvalues.

As an example, we use QR iteration on the matrix

$$\begin{bmatrix} 2.9766 & 0.3945 & 0.4198 & 1.1159 \\ 0.3945 & 2.7328 & -0.3097 & 0.1129 \\ 0.4198 & -0.3097 & 2.5675 & 0.6079 \\ 1.1159 & 0.1129 & 0.6079 & 1.7231 \end{bmatrix}$$

which has eigenvalues 1, 2, 3 and 4.

```
def qr_iter(A):
    """QR iteration"""
    for _ in range(10):
        q, r = linalg.qr(A)
        A = np.dot(r, q)
        print(A)
        print()
    return A


A = np.array(
    [
        [2.9766, 0.3945, 0.4198, 1.1159],
        [0.3945, 2.7328, -0.3097, 0.1129],
        [0.4198, -0.3097, 2.5675, 0.6079],
        [1.1159, 0.1129, 0.6079, 1.7231],
    ]
)
with np.printoptions(precision=2, suppress=True):
    qr_iter(A)
```

```
[[ 3.77  0.17  0.51 -0.39]
 [ 0.17  2.77 -0.39  0.05]
 [ 0.51 -0.39  2.4  -0.12]
 [-0.39  0.05 -0.12  1.06]]
```

(continues on next page)

(continued from previous page)

```
[[ 3.94  0.01  0.3   0.1 ]
 [ 0.01  2.87 -0.34 -0.03]
 [ 0.3  -0.34  2.18  0.01]
 [ 0.1  -0.03  0.01  1.  ]]

[[ 3.98 -0.04  0.16 -0.03]
 [-0.04  2.94 -0.24  0.01]
 [ 0.16 -0.24  2.07  0.  ]
 [-0.03  0.01  0.    1.  ]]

[[ 3.99 -0.04  0.08  0.01]
 [-0.04  2.97 -0.17 -0.  ]
 [ 0.08 -0.17  2.03 -0.  ]
 [ 0.01 -0.   -0.    1.  ]]

[[ 4.   -0.04  0.04 -0.  ]
 [-0.04  2.99 -0.11  0.  ]
 [ 0.04 -0.11  2.01  0.  ]
 [-0.    0.    0.    1.  ]]

[[ 4.   -0.03  0.02  0.  ]
 [-0.03  3.   -0.07 -0.  ]
 [ 0.02 -0.07  2.01 -0.  ]
 [ 0.   -0.   -0.    1.  ]]

[[ 4.   -0.02  0.01 -0.  ]
 [-0.02  3.   -0.05  0.  ]
 [ 0.01 -0.05  2.    0.  ]
 [-0.    0.    0.    1.  ]]

[[ 4.   -0.02  0.01  0.  ]
 [-0.02  3.   -0.03 -0.  ]
 [ 0.01 -0.03  2.   -0.  ]
 [ 0.   -0.   -0.    1.  ]]

[[ 4.   -0.01  0.   -0.  ]
 [-0.01  3.   -0.02  0.  ]
 [ 0.   -0.02  2.    0.  ]
 [-0.    0.    0.    1.  ]]

[[ 4.   -0.01  0.    0.  ]
 [-0.01  3.   -0.01 -0.  ]
 [ 0.   -0.01  2.   -0.  ]
 [ 0.   -0.   -0.    1.  ]]
```

To speed up this procedure we can use shifts, similar to their use in the power method. The most straightforward choice as shift is the lower right element of the matrix, but depending on the specifics of the problem better shifts might exist. In the example below, note how the obtained off-diagonal entries converge faster to zero than in the case without shifts.

```
def qr_iter_shift(A):
    """QR iteration with shift."""
    for _ in range(9):
        shift = A[len(A) - 1][len(A) - 1]
        q, r = linalg.qr(A - shift * np.identity(len(A)))
```

(continues on next page)

(continued from previous page)

```
        A = np.dot(r, q) + shift * np.identity(len(A))
        print(A)
        print()
    return A


A = np.array(
    [
        [2.9766, 0.3945, 0.4198, 1.1159],
        [0.3945, 2.7328, -0.3097, 0.1129],
        [0.4198, -0.3097, 2.5675, 0.6079],
        [1.1159, 0.1129, 0.6079, 1.7231],
    ]
)
with np.printoptions(precision=2, suppress=True):
    qr_iter_shift(A)
```

```
[[ 3.88 -0.02  0.24  0.51]
 [-0.02  2.95 -0.21 -0.16]
 [ 0.24 -0.21  2.04 -0.1 ]
 [ 0.51 -0.16 -0.1   1.13]]

[[ 3.99 -0.06  0.05  0.02]
 [-0.06  3.   -0.09 -0.01]
 [ 0.05 -0.09  2.01 -0.03]
 [ 0.02 -0.01 -0.03  1.  ]]

[[ 4.   -0.04  0.02  0.  ]
 [-0.04  3.   -0.04 -0.  ]
 [ 0.02 -0.04  2.   -0.  ]
 [ 0.   -0.   -0.    1.  ]]

[[ 4.   -0.03  0.01  0.  ]
 [-0.03  3.   -0.02 -0.  ]
 [ 0.01 -0.02  2.   -0.  ]
 [ 0.   -0.   -0.    1.  ]]

[[ 4.   -0.02  0.    0.  ]
 [-0.02  3.   -0.01 -0.  ]
 [ 0.   -0.01  2.    0.  ]
 [ 0.   -0.   -0.    1.  ]]

[[ 4.   -0.01  0.   -0.  ]
 [-0.01  3.   -0.01  0.  ]
 [ 0.   -0.01  2.   -0.  ]
 [-0.    0.    0.    1.  ]]

[[ 4.   -0.01  0.    0.  ]
 [-0.01  3.   -0.   -0.  ]
 [ 0.   -0.    2.    0.  ]
 [ 0.   -0.   -0.    1.  ]]

[[ 4.   -0.01  0.   -0.  ]
 [-0.01  3.   -0.    0.  ]
```

(continues on next page)

(continued from previous page)

```
 [ 0.   -0.    2.   -0.  ]
 [ 0.    0.    0.    1.  ]]

[[ 4. -0. -0.  0.]
 [-0.  3.  0. -0.]
 [-0.  0.  2. -0.]
 [ 0.  0.  0.  1.]]
```

4.3 Calculating the Singular Value Decomposition

Recall from the notebook about linear least squares that the **singular value decomposition (SVD)** of an $m \times n$ matrix $\mathbf{A}$ has the form

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\intercal$$

where $\mathbf{U}$ is an $m \times m$ orthogonal matrix, $\mathbf{V}$ is an $n \times n$ orthogonal matrix, and $\mathbf{\Sigma}$ is an $m \times n$ diagonal matrix, with

$$\sigma_{ij} = \begin{cases} 0, & \text{for } i \neq j \\ \sigma_i \geq 0, & \text{for } i = j \end{cases}$$

The diagonal entries σ_i are called the **singular values** of $\mathbf{A}$ and are usually ordered so that $\sigma_{i-1} \geq \sigma_i, i = 2, \ldots, \min\{m, n\}$, i.e. from largest value (upper left) to smallest value (bottom right). The columns $\mathbf{u}_i$ of $\mathbf{U}$ and $\mathbf{v}_i$ of $\mathbf{V}$ are the corresponding left and right **singular vectors**.

We discussed some handy applications of the SVD in that notebook, but postponed the calculation of the decomposition matrices. Here, we revisit the concept because singular values and vectors are intimately related to eigenvalues and eigenvectors. The singular values of $\mathbf{A}$ are the non-negative square roots of the eigenvalues of $\mathbf{A}^\intercal\mathbf{A}$, and the columns of $\mathbf{U}$ and $\mathbf{V}$ are orthonormal eigenvectors of $\mathbf{A}\mathbf{A}^\intercal$ and $\mathbf{A}^\intercal\mathbf{A}$, respectively.

> **Example**
>
> The singular value decomposition of the matrix
>
> $$\mathbf{A} = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix}$$

is given by

$$\mathbf{U}\mathbf{\Sigma}\mathbf{V}^\intercal = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 4 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}.$$

This statement can be verified by explicitly calculating $\mathbf{U}$, $\mathbf{\Sigma}$ and $\mathbf{V}$. We begin with

$$\mathbf{A}^\intercal\mathbf{A} = \mathbf{A}\mathbf{A}^\intercal = \begin{bmatrix} 10 & 6 \\ 6 & 10 \end{bmatrix}$$

which are equal here because A is a symmetric matrix. We can employ one of the methods discussed above to calculate its eigenvalues and eigenvectors. These are $\lambda_1 = \sigma_1^2 = 16$ with eigenvector $\mathbf{v}_1 = [1, 1]^\intercal$ and $\lambda_2 = \sigma_2^2 = 4$

with $\mathbf{v}_2 = [-1, 1]^\intercal$. The eigenvectors are easily converted to their orthonormal form, which results in

$$\mathbf{U} = \mathbf{V} = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}.$$

Now we construct $\mathbf{\Sigma}$ as $\text{diag}\left(\sqrt{\lambda_1}, \sqrt{\lambda_2}\right)$ and transpose $\mathbf{V}$ in order to find the proposed SVD.

```
# code for the example

A = np.array([(3, 1), (1, 3)])

transA = A.transpose()
product = A @ transA
eigenvalues, U = np.linalg.eig(product)
eigenvalues = np.sqrt(eigenvalues)
Delta = np.diag(eigenvalues)

SVD = U @ Delta @ U.transpose()

print("A^\intercalA = AA^\intercal = ")
print(product)
print("\neigenvalues =")
print(eigenvalues)

print("\nU = V =")
print(U)

print("\nSVD =")
print(SVD)
```

```
A^\intercalA = AA^\intercal =
[[10  6]
 [ 6 10]]

eigenvalues =
[4. 2.]

U = V =
[[ 0.70710678 -0.70710678]
 [ 0.70710678  0.70710678]]

SVD =
[[3. 1.]
 [1. 3.]]
```

```
# Or with SciPy's SVD
U, s, vh = linalg.svd(A)
SVD = U @ np.diag(s) @ vh
print(U, "\n\n", s, "\n\n", vh, "\n\n", SVD)
```

```
[[-0.70710678 -0.70710678]
 [-0.70710678  0.70710678]]
```

(continues on next page)

(continued from previous page)

```
 [4. 2.]

 [[-0.70710678 -0.70710678]
 [-0.70710678  0.70710678]]

 [[3. 1.]
 [1. 3.]]
```

4.4 Software

Until just a few years ago, QR iteration was the standard method for computing all of the eigenvalues (and optionally eigenvectors) of the resulting tridiagonal matrix. More recently, however, first divide-and-conquer and then relatively robust representation (RRR) methods (not shown in this course) have surpassed QR iteration in speed for computing all the eigenvectors. Implementations of both of these newer methods are available in LAPACK, but they do not yet have the decades-long record of reliability enjoyed by QR iteration. Thus, for now, the choice is between the speed of the newer methods, especially RRR, and the more proven dependability of QR iteration.

In `scipy` the most general method you can use is `linalg.eig`, which uses QR iteration. There exist similar methods to calculate an SVD. However, if your matrix has special properties, there are faster options that specifically make use of this information:

Method	Description
`eig`	Solve an ordinary or generalized eigenvalue problem of a square matrix.
`eigvals`	Compute eigenvalues from an ordinary or generalized eigenvalue problem.
`eigh`	Solve a standard or generalized eigenvalue problem for a complex Hermitian or real symmetric matrix.
`eigvalsh`	Solves a standard or generalized eigenvalue problem for a complex Hermitian or real symmetric matrix.
`eig_banded`	Solve real symmetric or complex Hermitian band matrix eigenvalue problem.
`eigvals_banded`	Solve real symmetric or complex Hermitian band matrix eigenvalue problem.
`eigh_tridiagonal`	Solve eigenvalue problem for a real symmetric tridiagonal matrix.
`eigvalsh_tridiagon`	Solve eigenvalue problem for a real symmetric tridiagonal matrix.
`svd`	Compute the single decomposition matrices.
`svdvals`	Compute singular values of a matrix.

Further documentation can be found here and here.

An example of the use of `linalg.eig` is shown below.

```
A = np.array(
    [
        [2.9766, 0.3945, 0.4198, 1.1159],
        [0.3945, 2.7328, -0.3097, 0.1129],
        [0.4198, -0.3097, 2.5675, 0.6079],
        [1.1159, 0.1129, 0.6079, 1.7231],
    ]
)
```

(continues on next page)

(continued from previous page)

```
la, v = linalg.eig(A)
l1, l2, l3, l4 = la
print(l1, l2, l3, l4)  # eigenvalues

print(v[:, 0])  # first eigenvector
print(v[:, 1])  # second eigenvector
print(v[:, 2])  # third eigenvector
print(v[:, 3])  # fourth eigenvector
```

```
(4.000021758462927+0j) (0.999983830092423+0j) (2.0000194591485454+0j) (2.
↪99997495229611+0j)
[0.76055983 0.18497226 0.38903703 0.48578204]
[ 0.44570011 -0.00622023  0.21627571 -0.8686412 ]
[ 0.4522908  -0.54893289 -0.70021691  0.06166026]
[-0.13540006 -0.81511916  0.55818571  0.07534114]
```

4.5 Physics Example: spring-and-mass system

> This example is partially inspired by the cc-licensed material from Michael Richmond found here.

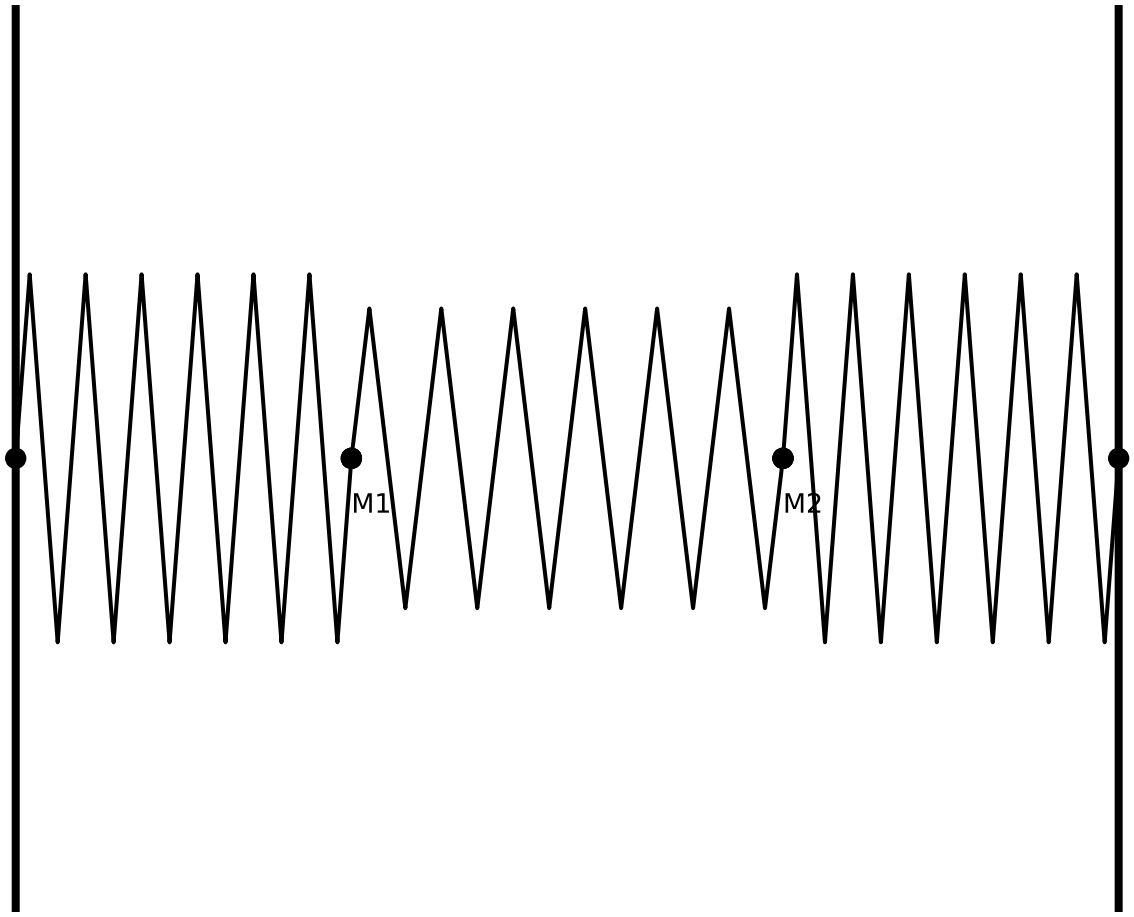
Consider the following system consisting of 2 masses, connected by identical springs fixed to a wall at the sides

```
def mk_spring(x1, x2, steps=12, height=1.0):
    l = x2 - x1
    nd = np.sqrt(max(0, height**2 - (l**2 / steps**2))) / 2

    def rx(i):
        return x1 + (l * (2 * i - 1)) / (2 * steps)

    def ry(i):
        return nd * (i % 2 * 2 - 1)

    s = [(rx(i), ry(i)) for i in range(1, steps + 1)]
    s = [(x1, 0), *s, (x2, 0)]

    return np.array(s)


def draw_springs(spring_configs):
    # The use of matplotlib to draw the spring system is considered
    # "spielerei" and not part of the course material.
    springs = [
        mk_spring(x1, x2, steps=s, height=h) for (x1, x2, s, h) in spring_
↪configs
    ]

    # Plot settings
    plt.close("springs")
    fig, ax = plt.subplots(num="springs")
    ax.axis("off")
    ax.set_ylim(-1, 1)
```

(continues on next page)

(continued from previous page)

```
    # Walls
    ax.axvline(x=0, color="k", linewidth=3)
    ax.axvline(x=23, color="k", linewidth=3)

    # Springs
    for spring in springs:
        ax.plot(*spring.T, "k")
        ax.plot(*spring[0], "ko", markersize=7)
        ax.plot(*spring[-1], "ko", markersize=7)

    # Labels
    for idx, spring in enumerate(springs[1:]):
        ax.plot(spring[0][0], spring[0][1], "ko", markersize=7)
        ax.text(spring[0][0], spring[0][1] - 0.12, f"M{idx+1}")


draw_springs([(0, 7, 12, 1), (7, 16, 12, 1), (16, 23, 12, 1)])
```



Let's call x_1 and x_2 the displacements of M_1 and M_2, respectively, from their equilibrium positions.

The forces (which define the accelerations) acting on each mass are

$$F_1 = M_1 \frac{d^2x_1}{dt^2} = -kx_1 + k(x_2 - x_1)$$
$$F_2 = M_2 \frac{d^2x_2}{dt^2} = -k(x_2 - x_1) - kx_2$$

This can be written as the following matrix equation

$$\begin{bmatrix} -\frac{2k}{M_1} & \frac{k}{M_1} \\ \frac{k}{M_2} & -\frac{2k}{M_2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \frac{d^2x_1}{dt^2} \\ \frac{d^2x_2}{dt^2} \end{bmatrix}$$

We're looking for a specific combination of x_1 and x_2 for which $\mathbf{Ax} = \lambda\mathbf{x}$ with λ the eigenvalue and $\mathbf{x}$= $[a\ b]^\mathsf{T}$ the corresponding eigenvector.

When comparing this to our original matrix equation, this means that

$$\begin{bmatrix} -\frac{2k}{M_1} & \frac{k}{M_1} \\ \frac{k}{M_2} & -\frac{2k}{M_2} \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \lambda \begin{bmatrix} a \\ b \end{bmatrix}$$

We can move the right-hand side to the left and end up with

$$\begin{bmatrix} -\frac{2k}{M_1} & \frac{k}{M_1} \\ \frac{k}{M_2} & -\frac{2k}{M_2} \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} - \lambda \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Writing these equations out explicitly gives

$$\left(\frac{-2k}{M_1} - \lambda\right) a + \frac{k}{M_1} b = 0$$
$$\frac{k}{M_2} a - \left(\frac{2k}{M_2} + \lambda\right) b = 0$$

The latter can be written as

$$a = \frac{M_2}{k}\left(\frac{2k}{M_2} + \lambda\right) b = \left(2 + \frac{M_2\lambda}{k}\right) b$$

and be filled in in the former to give

$$\left(\frac{-2k}{M_1} - \lambda\right)\left(2 + \frac{M_2\lambda}{k}\right) b + \frac{k}{M_1} b = 0$$
$$-\frac{M_2}{k}\lambda^2 - \left(2 + \frac{2M_2}{M_1}\right)\lambda - \frac{3k}{M_1} = 0$$

multiply by $-\frac{k}{M_2}$

$$\lambda^2 + 2k\left(\frac{M_1 + M_2}{M_1 M_2}\right)\lambda + \frac{3k^2}{M_1 M_2} = 0$$

Solving this for λ gives

$$\lambda = \frac{-2k\left(\frac{M_1+M_2}{M_1M_2}\right) \pm \sqrt{4k^2\left(\frac{M_1+M_2}{M_1M_2}\right)^2 - 4\frac{3k^2}{M_1M_2}}}{2}$$

Or,

$$\lambda = \frac{-k}{M_1 M_2}\left[(M_1 + M_2) \pm \sqrt{(M_1 - M_2)^2 + M_1 M_2}\right]$$

For simplicity, let's assume that $M_1 = M_2$ so this reduces to $\lambda = -k/M$ and $\lambda = -3k/M$.

Using these λ in the following set of equations

$$\left(\frac{-2k}{M_1} - \lambda\right) a + \frac{k}{M_1} b = 0$$
$$\frac{k}{M_2} a - \left(\frac{2k}{M_2} + \lambda\right) b = 0$$

gives

$$\left(\frac{-2k}{M} + \frac{k}{M}\right) a + \frac{k}{M} b = 0$$
$$\frac{k}{M} a - \left(\frac{2k}{M} - \frac{k}{M}\right) b = 0$$

and

$$\left(\frac{-2k}{M} + 3\frac{k}{M}\right) a + \frac{k}{M} b = 0$$
$$\frac{k}{M} a - \left(\frac{2k}{M} - 3\frac{k}{M}\right) b = 0$$

which can be solved as $a = b$ and $a = -b$, so the corresponding eigenvectors are $[1\ 1]^\intercal$ and $[1\ -1]^\intercal$

We could have saved ourselves all this work if we would just have asked scipy:

For instance, for $k = M_1 = M_2 = 1$, we would find

```
A = np.array([(-2, 1), (1, -2)])
```

```
la, v = linalg.eig(A)
print(la, "\n", v)
```

```
[-1.+0.j -3.+0.j]
 [[ 0.70710678 -0.70710678]
 [ 0.70710678  0.70710678]]
```

Which are indeed the expected eigenvalues of -1 and -3 and eigenvectors $[1\ 1]^\intercal$ and $[1\ -1]^\intercal$ (normalized to 1)

> You could find a simple animation for the three eigenmodes of the oscillator here. The animations and text on the page are ©2004-2013 by Daniel A. Russell.

If we now look at this combination of the original equations in our set

$$F_1 = M\frac{d^2x_1}{dt^2} = -kx_1 + k(x_2 - x_1)$$
$$F_2 = M\frac{d^2x_2}{dt^2} = -k(x_2 - x_1) - kx_2$$

we find for the eigenvector $[11]^\intercal$:

$$M\frac{d^2x_1}{dt^2} + M\frac{d^2x_1}{dt^2} = -k(x_1 + x_2)$$

and for eigenvector $[1\ -\ 1]^\intercal$:

$$M\frac{d^2x_2}{dt^2} - M\frac{d^2x_1}{dt^2} = -3k(x_2 - x_1)$$

when introducing the variables $s_1 = (x_1 + x_2)$ and $s_2 = (x_2 - x_1)$ this results in the following equations

$$\frac{d^2 s_1}{dt^2} = -\frac{k}{M} s_1$$
$$\frac{d^2 s_2}{dt^2} = -\frac{3k}{M} s_2$$

which can easily be solved as

$$x_1 + x_2 = s_1 = A \cos\left(\sqrt{k/M}t + \phi\right)$$
$$x_2 - x_1 = s_2 = A \cos\left(\sqrt{3k/M}t + \phi\right)$$

which shows that solving $\boldsymbol{A}\mathbf{s} = \lambda\mathbf{s}$ very elegantly give you the dynamical equations that describe this system.

4.5.1 Dynamical problem

Let's say we would want to know the position of the first block at time $t = 5s$, given the following initial conditions.

- the mass equals $M = 1\,\text{kg}$
- the constant of the springs $k = 1\,\text{N/m}$
- at $t = 0$, the first mass is at position $x_1 = 2\,\text{m}$
- at $t = 0$ the second mass is at position $x_2 = -1\,\text{m}$
- at $t = 0$, the starting velocity is $v_1 = -1\,\text{m/s}$
- at $t = 0$, the staring velocity is $v_2 = 1\,\text{m/s}$

We can find the constants of integration by plugging these conditions into s_1 and s_2.

$$s_1(0) = 2 - 1 = A_1 \cos\left(\sqrt{1/1} \cdot 0 + \phi_1\right)$$

$$\implies \begin{cases} A_1 = 1\,\text{m} \\ \phi_1 = 0 \end{cases}$$

By using the same method in s_2 and its time derivative, we find;

$$\begin{cases} A_2 = -3.21\,\text{m} \\ \phi_2 = 0.364\,\text{rad} \end{cases}$$

Such that,

$$s_1(t) = (1\,\text{m}) \cos\left(\omega_1 t\right)$$
$$s_2(t) = (-3.21\,\text{m}) \cos\left(\omega_2 t + 0.364\,\text{rad}\right)$$

We want to know the positions of the actual boxes, so we need the expressions for $x_1(t)$ and $x_2(t)$ instead of $s_1(t)$ and $s_2(t)$.

$$\begin{aligned} x_1(t) &= \frac{1}{2}\left[s_1(t) - s_2(t)\right] \\ &= (0.5\,\text{m}) \cos(\omega_1 t) + (1.61\,\text{m}) \cos(\omega_2 t + 0.364\,\text{rad}) \end{aligned}$$

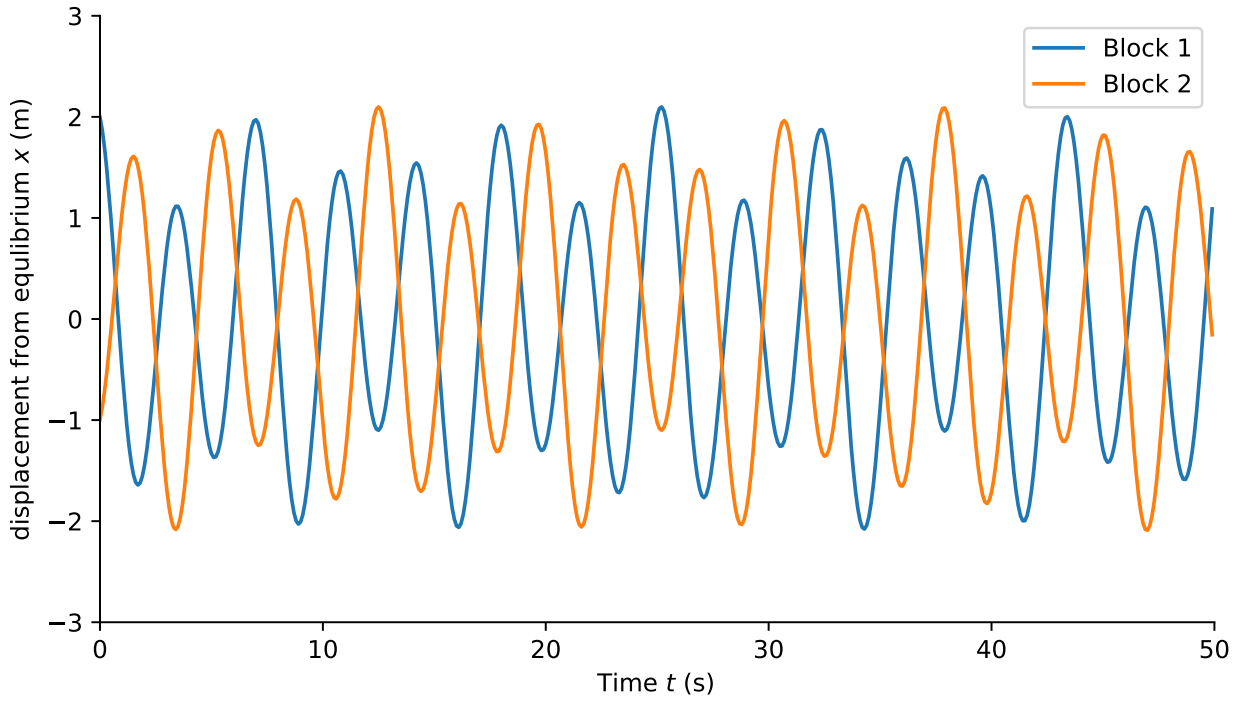
$$\begin{aligned} x_2(t) &= \frac{1}{2}\left[s_1(t) + s_2(t)\right] \\ &= (0.5\,\text{m}) \cos(\omega_1 t) - (1.61\,\text{m}) \cos(\omega_2 t + 0.364\,\text{rad}) \end{aligned}$$

To find the positions of the masses we define the functions found with the initial conditions:

```
def x1(t):
    k = 1
    M = 1
    pos = 0.5 * np.cos(np.sqrt(k / M) * t) + 1.6 * np.cos(
        np.sqrt(3 * k / M) * t + 0.364
    )
    return pos


def x2(t):
    k = 1
    M = 1
    pos = 0.5 * np.cos(np.sqrt(k / M) * t) - 1.6 * np.cos(
        np.sqrt(3 * k / M) * t + 0.364
    )
    return pos


print("Mass one has position x =", x1(5), "at time t = 5s.")
print("Mass two has position x =", x2(5), "at time t = 5s.")
```

```
Mass one has position x = -1.331539855111344 at time t = 5s.
Mass two has position x = 1.6152020405745702 at time t = 5s.
```

```
def plot_solution():
    """The positions of the masses over time for our initial conditions."""
    plt.close("solution")
    fig, ax = plt.subplots(figsize=(7, 4), num="solution")
    t = np.arange(0, 50, 0.1)
    ax.plot(t, x1(t))
    ax.plot(t, x2(t))

    ax.set_xlim([0, 50])
    ax.set_ylim([-3, 3])
    ax.legend(("Block 1", "Block 2"))
    ax.set_xlabel("Time $t$ (s)")
    ax.set_ylabel("displacement from equlibrium $x$ (m)")


plot_solution()
```



Alternative solution

Let's solve the same problem by integrating the differential equations (see notebook integrating ODE's).

Adapted from https://scipy-cookbook.readthedocs.io/items/CoupledSpringMassSystem.html

```
def vectorfield(w, t, p):
    """Defines the differential equations for the coupled spring-mass system.

    Parameters
    ----------
    w
        vector of the state variables: w = [x1,y1,x2,y2].
    t
        time.
    p
        vector of the parameters: p = [m1,m2,k1,k2].
    """
    x1, y1, x2, y2 = w
    m1, m2, k1, k2 = p

    # Create f = (x1',y1',x2',y2'):
    f = [y1, (-k1 * x1 + k2 * (x2 - x1)) / m1, y2, (-k2 * (x2 - x1) - k2 *␣
↪x2) / m2]
    return f
```

The code above explicitly implements the following set of 1st order differential equations, equivalent to the 2 2nd order

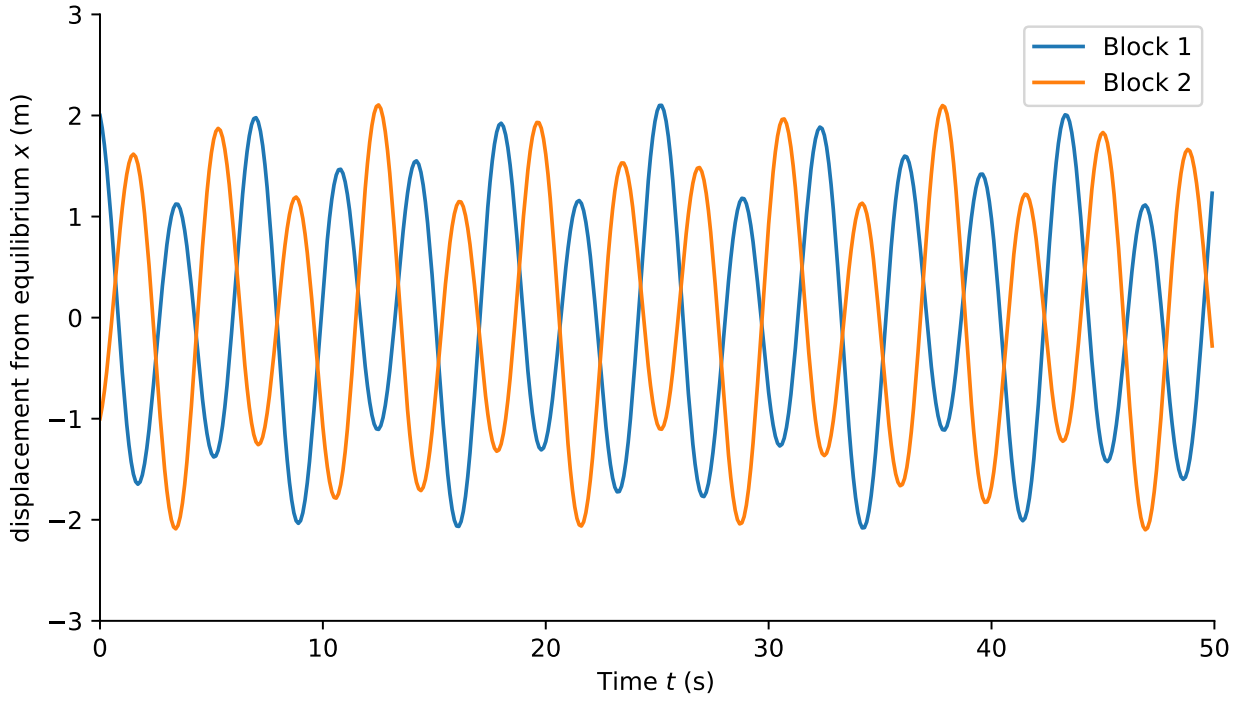
ODE's given by Newton's law we initially began the example with:

$$\begin{cases} \frac{dx_1}{dt} = y_1 \\ \frac{dy_1}{dt} = \frac{-k_1 x_1 + k_2(x_2 - x_1)}{m_1} \\ \frac{dx_2}{dt} = y_2 \\ \frac{dy_2}{dt} = \frac{-k_2(x_2 - x_1) - k_2 x_2}{m_2} \end{cases}$$

Below, it will be solved using methods we saw in the ODE notebook.

```
def solve_ode():
    """Use ODEINT to solve the differential equations defined by the vector␣
 ↪field."""

    # Parameter values
    # Masses:
    m1 = 1.0
    m2 = 1.0
    # Spring constants
    k1 = 1
    k2 = 1

    # Initial conditions
    # x1 and x2 are the initial displacements; y1 and y2 are the initial␣
 ↪velocities
    x1 = 2.0
    y1 = -1.0
    x2 = -1.0
    y2 = 1.0

    # ODE solver parameters
    abserr = 1.0e-8
    relerr = 1.0e-6
    stoptime = 50.0
    numpoints = 1000

    # Create the time samples for the output of the ODE solver.
    # I use a large number of points, only because I want to make
    # a plot of the solution that looks nice.
    t = [stoptime * float(i) / (numpoints - 1) for i in range(numpoints)]

    # Pack up the parameters and initial conditions:
    p = [m1, m2, k1, k2]
    w0 = [x1, y1, x2, y2]

    # Call the ODE solver.
    return integrate.odeint(vectorfield, w0, t, args=(p,), atol=abserr,␣
 ↪rtol=relerr)


wsol = solve_ode()
```

```
def plot_alternative_solution():
    plt.close("alternative")
    fig, ax = plt.subplots(figsize=(7, 4), num="alternative")

    t = np.arange(0, 50, 0.1)
    ax.plot(t, wsol[::2, 0])
    ax.plot(t, wsol[::2, 2])

    ax.set_xlim([0, 50])
    ax.set_ylim([-3, 3])
    ax.legend(("Block 1", "Block 2"))
    ax.set_xlabel("Time $t$ (s)")
    ax.set_ylabel("displacement from equilibrium $x$ (m)")


plot_alternative_solution()
```



CHAPTER 5

Nonlinear equations

```
import matplotlib.pyplot as plt
import numpy as np
import numpy.polynomial.polynomial as poly
from matplotlib.animation import FuncAnimation
from scipy import optimize
```

5.1 Introduction

Although many phenomena in nature can be described (or approximated) by a linear response, many others are inherently **nonlinear** in that effects are not directly proportional to their causes. For instance, the air resistance of a moving object is proportional to the square of the velocity.

In analogy to linear equations, where a system of equations is written as $\mathbf{Ax} = \mathbf{b}$, we could write down a system of nonlinear equations as $\mathbf{f}(\mathbf{x}) = \mathbf{y}$.

However, it is more customary to subtract $\mathbf{y}$ from $\mathbf{f}(\mathbf{x})$ so the equation that needs to be solved is expressed as $\mathbf{f}(\mathbf{x}) = \mathbf{0}$.

In one dimension, this means that we are looking for the intersection of a curve with the x-axis. In general, we seek a vector $\mathbf{x}$ such that all component function $\mathbf{f}(\mathbf{x})$ are zero simultaneously.

A solution value $\mathbf{x}$ such that $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ is called a **root** of the equation, and a **zero** of the function $\mathbf{f}$. This problem is thus referred to as **root finding** or **zero finding**.

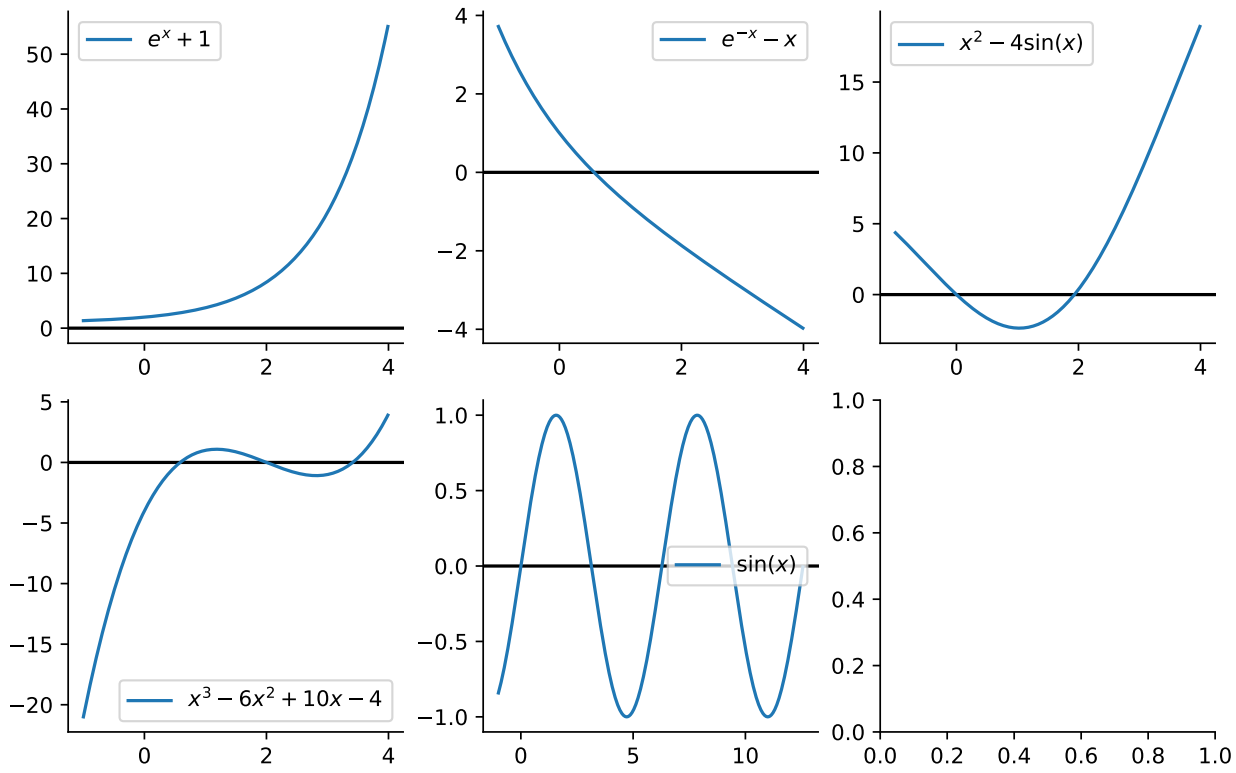
5.2 Number of solutions

For linear systems, each equation describes a *flat* hyperplane in $\mathbb{R}^n$, and the solution corresponded with the points where all of them intersect. For nonlinear equations, this is also true, but here each equation can describe a *curved* hyperplane. Because curved surfaces can intersect in many more ways than flat ones, it is not possible to make general statements about the number of solutions to a nonlinear problem.

> **Examples**
>
> Even in 1 dimension, many different cases are possible:

- $e^x + 1 = 0$ has no solution.
- $e^{-x} - x = 0$ has one solution.
- $x^2 - 4\sin(x) = 0$ has two solutions.
- $x^3 - 6x^2 + 10x - 4 = 0$ has three solutions.
- $\sin(x) = 0$ has infinitely many solutions.

```
def fig_nr_solutions():
    plt.close("nr_solutions")
    fig, axs = plt.subplots(2, 3, figsize=(8, 5), num="nr_solutions")
    x = np.arange(-1, 4, 0.01)

    plots = [
        np.exp(x) + 1,
        np.exp(-x) - x,
        x**2 - 4 * np.sin(x),
        x**3 - 6 * x**2 + 10 * x - 4,
        np.sin(np.arange(-1, 4 * np.pi, 0.01)),
    ]
    labels = [
        "$e^x+1$",
        "$e^{-x}-x$",
        r"$x^2 - 4 \sin(x)$",
        "$x^3-6x^2+10x-4$",
        r"$\sin(x)$",
    ]

    for i in range(5):
        row, col = divmod(i, 3)
        ax = axs[row, col]

        if i == 4:
            x = np.arange(-1, 4 * np.pi, 0.01)

        ax.axhline(0, color="black")
        ax.plot(x, plots[i], label=labels[i])
        ax.legend()


fig_nr_solutions()
```



For a nonlinear equation it is possible to have degenerate solutions, which are called **multiple roots**. Generally, for a smooth function f, if $f(x^*) = f'(x^*) = f''(x^*) = \cdots = f^{(m-1)}(x^*) = 0$ and $f^m(x^*) \neq 0$, then x^* is a root of **multiplicity** m.

If $m = 1$, then the solution is not degenerate and is called a **simple root**.

Geometrically, this means that the curve defined by f has a horizontal tangent at the x-axis.

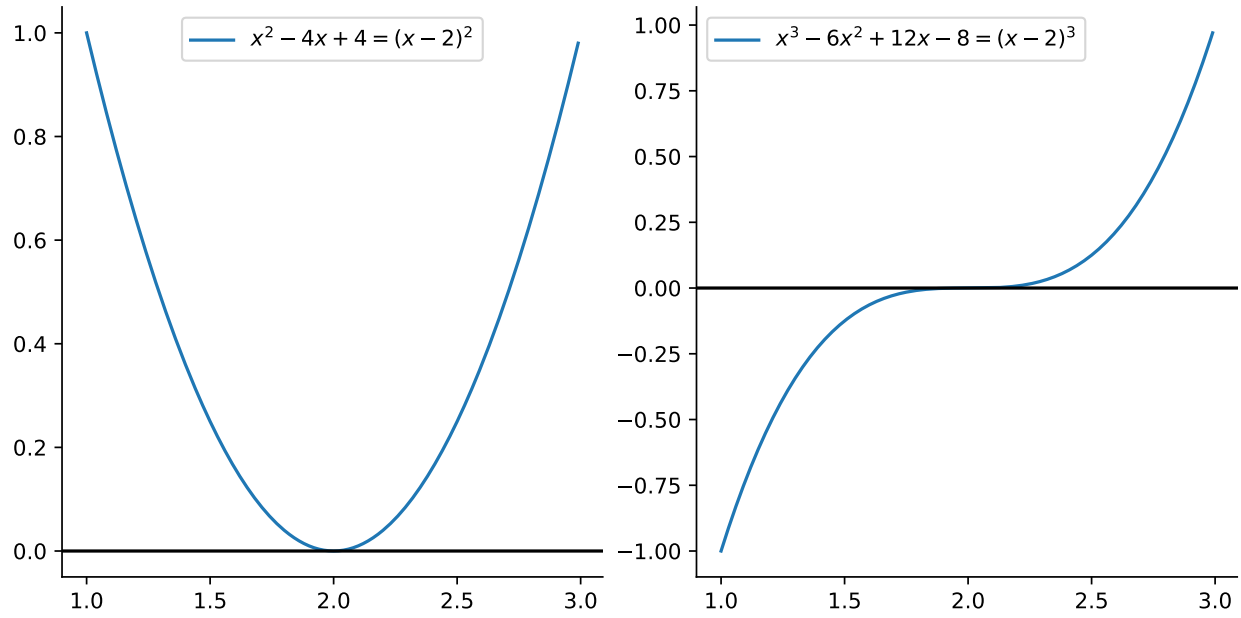
> **Examples**
>
> $x^2 - 4x + 4 = (x-2)^2 = 0$ has a root $x = 2$ of multiplicity 2
>
> $x^3 - 6x^2 + 12x - 8 = (x-2)^3 = 0$ has a root $x = 2$ of multiplicity 3

```
def fig_multiplicity():
    plt.close("multiplicity")
    fig, axs = plt.subplots(1, 2, figsize=(8, 4), num="multiplicity")

    x = np.arange(1, 3, 0.01)

    axs[0].plot(x, x**2 - 4 * x + 4, label="$x^2-4x+4=(x-2)^2$")
    axs[0].axhline(0, color="black")
    axs[0].legend()

    axs[1].plot(
        x,
        x**3 - 6 * x**2 + 12 * x - 8,
        label="$x^3-6x^2+12x-8=(x-2)^3$",
    )
    axs[1].axhline(0, color="black")
```

(continues on next page)

(continued from previous page)

```
    axs[1].legend()


fig_multiplicity()
```



5.3 Sensitivity

Let's investigate the sensitivity of the root finding problem $f(x) = 0$, i.e. if x^* is a root of f, how much does x^* change for small changes to the parameters of f?

In one dimension, the condition number for the root-finding problem of f near x^* is $\frac{1}{\|f'(x^*)\|}$.

In other words, for functions for which $f'(x)$ is small near the root, the error in the root finding problem can be substantial.

At a multiple root x^*, $f'(x^*) = 0$, so the condition number of a multiple root is infinite. Intuitively this is clear because a small change in the parameters of f can cause the multiple root to disappear or split up in more than one root.

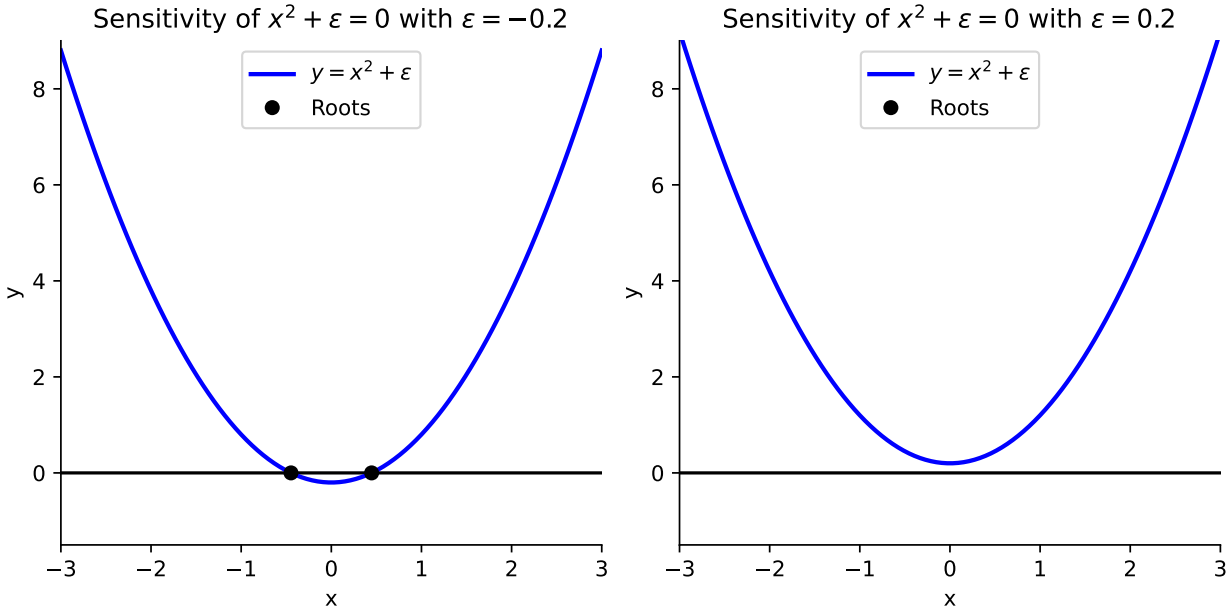
> **Example**
>
> As an example, consider the root-finding problem $f(x) = x^2 = 0$, which has twofold degenerate solution $x^* = 0$.
>
> For a small change $\epsilon > 0$ in f, we can find
>
> $x^2 - \epsilon = 0$, which has two roots at $\pm\sqrt{\epsilon}$
>
> or
>
> $x^2 + \epsilon = 0$, which has no roots

```
def plot_parabola(epsilon, ax):
    # Generate x data
    x = np.linspace(-3, 3, 100)
```

(continues on next page)

(continued from previous page)

```
    # Calculate the roots of the equation y = x^2 + epsilon = 0
    if epsilon < 0:
        roots = [np.sqrt(-epsilon), -np.sqrt(-epsilon)]
    elif epsilon == 0:
        roots = [0]
    else:
        roots = []

    # Plot the parabola and its roots
    ax.axhline(y=0, color="black")
    ax.plot(x, x**2 + epsilon, "b", linewidth=2.0, label=r"$y = x^2 + \epsilon
↪$")
    ax.plot(roots, [0 for _ in roots], "o", color="black", label="Roots")

    # Labeling
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_xlim(-3, 3)
    ax.set_ylim(-1.5, 9)
    ax.set_title(
        rf"Sensitivity of $x^2 + \epsilon = 0$ with $\epsilon = {epsilon:.1f}$
↪"
    )
    ax.legend()


# Set up the figure and subplots
plt.close("parabolas1")
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4), num="parabolas1")

# Plot for epsilon = -0.2
plot_parabola(-0.2, ax1)

# Plot for epsilon = 0.2
plot_parabola(0.2, ax2)
```



In multiple dimensions, the condition number is generalized using the Jacobian $\mathbf{J}$ to $\|\mathbf{J}^{-1}_{\mathbf{f}(\mathbf{x})}\|$.

Example

Consider the two-dimensional system

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} x_1^2 - x_2 + \gamma \\ -x_1 + x_2^2 + \gamma \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Each of these equations defines a parabola, and any point where they intersect is a solution to the system. Depending on the value of γ, this system can have either zero, one, two or four solutions.

For the specific case of $\gamma = 0.25$, both parabola touch each other, i.e. they have one degenerate solution.

For this value, the Jacobian matrix reads

$$\mathbf{J}_{\mathbf{f}(\mathbf{x})} = \begin{bmatrix} 2x_1 & -1 \\ -1 & 2x_2 \end{bmatrix}$$

which is singular at the unique solution $\mathbf{x}^* = [0.5, 0.5]^\intercal$.

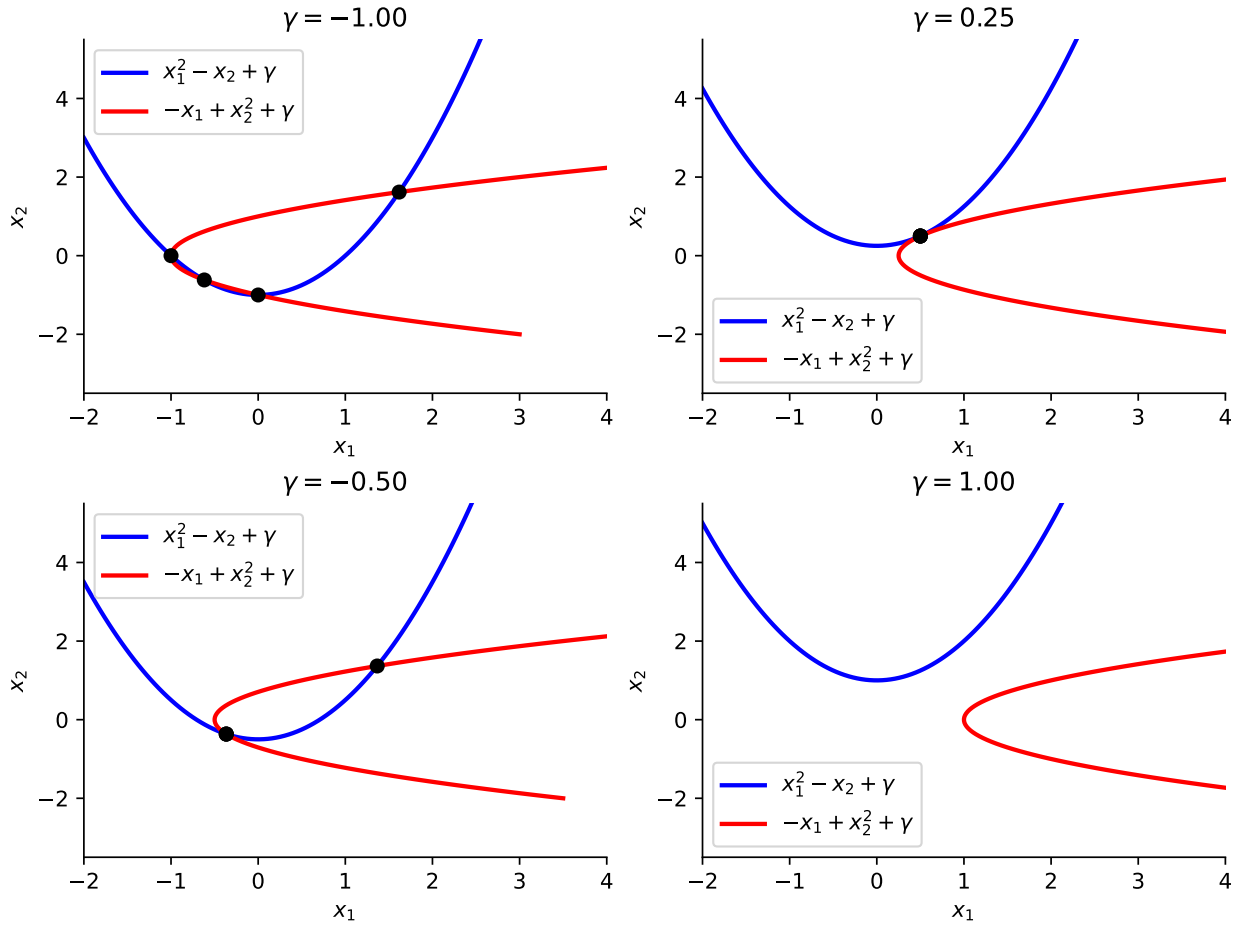
For a larger value of γ the parabola no longer intersect, and for a smaller value of γ, they intersect at 2 points.

```
def plot_parabolas(gamma, ax):
    # Define the x ranges for each curve
    x1, x2 = np.linspace(-2, 2.75, 100), np.linspace(-2, 2.75, 100)

    # Define the system of equations
    def f(x):
        return [x[0] ** 2 - x[1] + gamma, -x[0] + x[1] ** 2 + gamma]

    # Find intersection points
    roots = []
    start_conditions = np.array([[0, 0], [1, 1], [-1, 0], [0, -1]])
    for start_condition in start_conditions:
```

(continues on next page)

(continued from previous page)

```
        result = optimize.root(f, start_condition)
        if result.success:
            roots.append(result.x)

    # Plot the curves
    ax.plot(x1, x1**2 + gamma, "b", linewidth=2.0, label=r"$x_{1}^{2}-x_{2}+\
↪gamma$")
    ax.plot(
        x2**2 + gamma, x2, "r", linewidth=2.0, label=r"$-x_{1}+x_{2}^{2}+\
↪gamma$"
    )

    # Plot intersection points
    ax.plot(
        [root[0] for root in roots], [root[1] for root in roots], "o", color=
↪"black"
    )

    # Labeling and limits
    ax.set_xlabel("$x_{1}$")
    ax.set_ylabel("$x_{2}$")
    ax.set_title(rf"$\gamma = {gamma:.2f}$")
    ax.set_xlim(-2, 4)
    ax.set_ylim(-3.5, 5.5)
    ax.legend()


# Set up figure with subplots for each case
plt.close("parabolas2")
fig, axs = plt.subplots(2, 2, figsize=(8, 6), num="parabolas2")
gammas = [-1, 0.25, -0.5, 1]  # Example values for gamma with 0, 1, 2, and 4␣
↪roots

# Plot each case on a separate subplot
for ax, gamma in zip(axs.flat, gammas, strict=True):
    plot_parabolas(gamma, ax)
```



5.4 Convergence Rates and Stopping Criteria

The **convergence rate** is the effectiveness with which a certain algorithm reaches its solution.

To solve a nonlinear equation, one often has the choice between several iterative methods, with different converge rates. The total cost of solving the system does not only depend on the amount of iterations necessary to reach the solution with the desired accuracy, but also the computational complexity of a single iteration.

The convergence rate can be defined as follows:

Let $\mathbf{e}_k = \mathbf{x}_k - x^*$ be the error at iteration k, where $\mathbf{x}_k$ is the approximate solution at iteration k and x^* the (usually unknown) true solution.

An iterative method is said to converge with rate r if

$$\lim_{k \to \infty} \frac{\|\mathbf{e}_{k+1}\|}{\|\mathbf{e}_k\|^r} = C$$

for some finite constant $C > 0$.

Interesting cases are:

- $r = 1$ and $C < 1$: *linear* convergence
- $r > 1$: *superlinear* convergence
- $r = 2$: *quadratic* convergence
- $r = 3$: *cubic* convergence

In an iterative method with linear convergence, the solution gains an additional $-r \log_{10}(C)$ number of correct digits as compared to the previous iteration. For superlinearly convergent methods, the solution has about r times as many correct digits as compared to the previous iteration.

Example

To make this more concrete we will look at a couple of examples.

consider the following sequence.

$$\{1; 0.5; 0.25; 0.125; ...\}$$

We see that this sequence will converge to 0 so that we can define the errors as $\mathbf{e}_k = \mathbf{x}_k - x^* = \frac{1}{2^k} - 0$. This sequence has a linear convergence rate with $C = 0.5$. this is easily verified because we recognize the sequence of errors as $e_k = 1/2^k$

$$\lim_{k \to \infty} \frac{\frac{1}{2^{k+1}}}{\frac{1}{2^k}} = 0.5$$

Now consider a sequence of errors. This is the sequence of errors from the demonstration of Newton's method below (the errors are those of iterations 12-16, because then the values get close enough to the real value to be meaningful).

$$\{0.122; 0.0128; 0.00016; 2.66 \cdot 10^{-8}; 6.66 \cdot 10^{-16}\}$$

As we can see we get double the amount of precision each iteration, which means that we have a convergence rate of 2, i.e. quadratic convergence.

With this convergence rate of $r = 2$ we can calculate the constant C. This constant seems to get closer and closer to $C = 0.6$ In the last iteration we hit machine precision so we will not count that as a valid estimation for the value of C.

The convergence of a certain algorithm tells us that we zoom in on the correct solution at a certain rate, but it doesn't tell us the current accuracy of our solution at any given iteration.

Therefore, we don't know whether we reached a solution that is sufficiently close to the real solution to decide that we can stop the algorithm.

More often than not, it's not trivial to define a suitable **stopping criterion**. A reasonable way is to look at the relative change in the solutions for successive iterations $\|\mathbf{x}_{k+1} - \mathbf{x}_k\|/\|\mathbf{x}_k\| < \varepsilon$, and check that this quantity becomes smaller than a predefined **error tolerance** ε.

A sensible value for ε *might* be (but this really depends on your specific problem) the double precision accuracy of 10^{-16}.

5.5 Solving nonlinear equations in one dimension

Let's focus on how to find the solution to nonlinear equations in one dimension: For a continuous function $f : \mathbb{R} \to \mathbb{R}$, we seek a point $x^* \in \mathbb{R}$ such that $f(x^*) = 0$.

5.5.1 Bisection method

Because there might not exist a machine number x^* for which $f(x^*)$ is exactly zero using finite-precision arithmetic. An alternative is to look for a (short) interval $[a, b]$ in which f changes sign. Such a **bracket** ensures that the function must take a zero value somewhere within this interval.

The **bisection method** begins with an initial bracket and then iteratively reduces its length until the desired accuracy is

reached.

At every iteration, the function is evaluated at the **midpoint** of the interval, such that half of the interval can be discarded, based on the sign of the function value at the midpoint.

Convergence

The bisection method makes no use of the magnitudes of the function values, and as a result it is certain to converge, but very slowly. At each iteration, the bound on the possible error is reduced by half, meaning that it converges linearly with $r = 1$ and $C = 0.5$.

Given a starting interval $[a, b]$, the length of the interval after k iterations is $(b-a)/2^k$, so that achieving an error tolerance of ε requires n iterations, where

$$n = \log_2\left(\frac{b-a}{\varepsilon}\right) \qquad \Longleftrightarrow \qquad \varepsilon = \left(\frac{b-a}{2^n}\right)$$

regardless of the particular function f involved.

```
def func_bm(x):
    """Example function to demonstrate the bisection method."""
    return x * x - 4 * np.sin(x)


def bisection_method(f, a, b, tol):
    """Bisection method implementation to find the root
    of a function within an interval."""
    brackets = []
    while (b - a) >= tol:
        m = a + (b - a) / 2
        if np.sign(f(a)) == np.sign(f(m)):
            a = m
        else:
            b = m
        brackets.append([a, b, m])
        print(f"{len(brackets):3d}  {a:17.15f}  {b:17.15f}")
    return np.array(brackets)


def plot_bisection_with_function(f, a=1, b=6, tol=1e-8):
    # Perform bisection method and capture intervals
    results = bisection_method(f, a, b, tol)
    num_iterations = len(results)

    # Set up figure and subplots with shared x-axis
    plt.close("bisection")
    fig, (ax1, ax2) = plt.subplots(
        2,
        1,
        figsize=(8, 6),
        sharex=True,
        gridspec_kw={"height_ratios": [2, 1]},
        num="bisection",
    )

    # Top Plot: Function plot with zero crossing
    x = np.linspace(a, b, 400)
    y = f(x)
```

(continues on next page)

(continued from previous page)

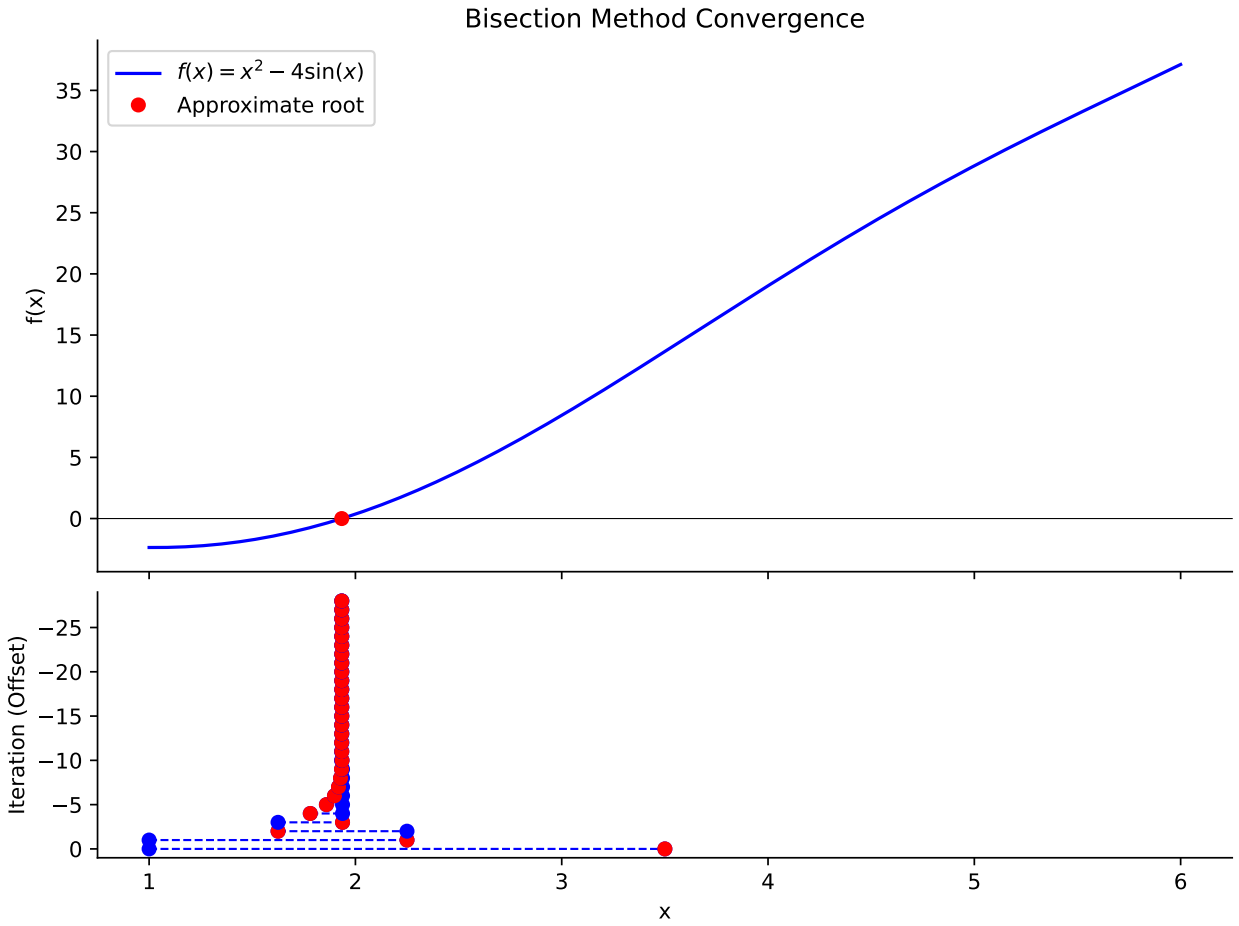
```
    ax1.plot(x, y, label=r"$f(x) = x^2 - 4\sin(x)$", color="blue")
    ax1.axhline(0, color="black", linewidth=0.5)
    ax1.set_ylabel("f(x)")
    ax1.set_title("Bisection Method Convergence")
    ax1.legend()

    # Highlight zero crossing (root) on the function plot, if known
    root_approx = results[-1, 2]  # Final midpoint as the root approximation
    ax1.plot(root_approx, f(root_approx), "ro", label="Approximate root")
    ax1.legend()

    # Bottom Plot: Bisection funnel
    for i, (left, right, mid) in enumerate(results):
        offset = -i  # Vertical offset for each interval
        ax2.hlines(
            offset, left, right, color="blue", linestyle="--", linewidth=1
        )  # Interval line
        ax2.plot([left, right], [offset, offset], "bo")  # Interval endpoints
        ax2.plot(mid, offset, "ro")  # Midpoint at each step

    # Bottom plot settings
    ax2.set_xlabel("x")
    ax2.set_ylabel("Iteration (Offset)")
    ax2.set_ylim(-num_iterations, 1)
    ax2.invert_yaxis()  # Invert y-axis for funnel effect


# Call the function to display the plot
plot_bisection_with_function(func_bm)
```

```
  1  1.000000000000000  3.500000000000000
  2  1.000000000000000  2.250000000000000
  3  1.625000000000000  2.250000000000000
  4  1.625000000000000  1.937500000000000
  5  1.781250000000000  1.937500000000000
  6  1.859375000000000  1.937500000000000
  7  1.898437500000000  1.937500000000000
  8  1.917968750000000  1.937500000000000
  9  1.927734375000000  1.937500000000000
 10  1.932617187500000  1.937500000000000
 11  1.932617187500000  1.935058593750000
 12  1.932617187500000  1.933837890625000
 13  1.933227539062500  1.933837890625000
 14  1.933532714843750  1.933837890625000
 15  1.933685302734375  1.933837890625000
 16  1.933685302734375  1.933761596679688
 17  1.933723449707031  1.933761596679688
 18  1.933742523193359  1.933761596679688
 19  1.933752059936523  1.933761596679688
 20  1.933752059936523  1.933756828308105
 21  1.933752059936523  1.933754444122314
 22  1.933753252029419  1.933754444122314
 23  1.933753252029419  1.933753848075867
 24  1.933753550052643  1.933753848075867
```

(continues on next page)

(continued from previous page)

```
25  1.933753699064255  1.933753848075867
26  1.933753699064255  1.933753773570061
27  1.933753736317158  1.933753773570061
28  1.933753754943609  1.933753773570061
29  1.933753754943609  1.933753764256835
```



The bisection method is also implemented in SciPy:

```
optimize.root_scalar(
    func_bm, method="bisect", bracket=[1, 6], xtol=1e-16, maxiter=200
)
```

```
      converged: True
           flag: converged
 function_calls: 54
     iterations: 52
           root: 1.933753762827022
         method: bisect
```

5.5.2 Fixed-point iteration

Let's now consider an alternative problem. Given a function $g : \mathbb{r} \to \mathbb{R}$, a value x such that $x = g(x)$ is called a **fixed point** of the function g, since x remains unchanged when g is applied to it.

Geometrically, finding such a fixed point corresponds to finding an intersection between g and the diagonal line $y = x$.

This problem is important because many iterative algorithms for solving nonlinear equations (see below) are based on iterations of the form

$$x_{k+1} = g(x_k)$$

where g is a function chosen so that its fixed points are solutions for $f(x) = 0$. Such a scheme is called **fixed-point iteration** or **functional iteration**, since the function g is applied repeatedly to an initial starting value x_0.

For a given function $f(x) = 0$, there are many equivalent fixed-point problems $x = g(x)$ with different choices for g. However, they are not all equally useful, as they may differ in their convergence rate and even whether or not they converge at all.

```
def func1(x):
    """Divergent function for fixed-point iteration."""
    return x**2 - 2


def func2(x):
    """Convergent function for fixed-point iteration."""
    return np.sqrt(x + 2)


def func3(x):
    """Convergent function for fixed-point iteration."""
    return 1 + 2 / x


def func4(x):
    """Function with potential divergence due to singularity."""
    with np.errstate(divide="ignore", invalid="ignore"):
        result = (x**2 + 2) / (2 * x - 1)
        if np.isscalar(result):
            return np.nan if np.isinf(result) else result
        result[np.isinf(result)] = np.nan
    return result


def fixed_point_iteration(f, x0, max_iter=10):
    """Perform fixed-point iteration, returning intermediate points."""
    results = [x0]
    for _ in range(max_iter):
        x_new = f(x0)
        if np.isnan(x_new):  # Stop if we encounter a singularity
            break
        results.extend([x0, x_new])  # Alternate: projection and function␣
↪evaluation
        x0 = x_new
    return results


def plot_fixed_point_iterations():
```

(continues on next page)

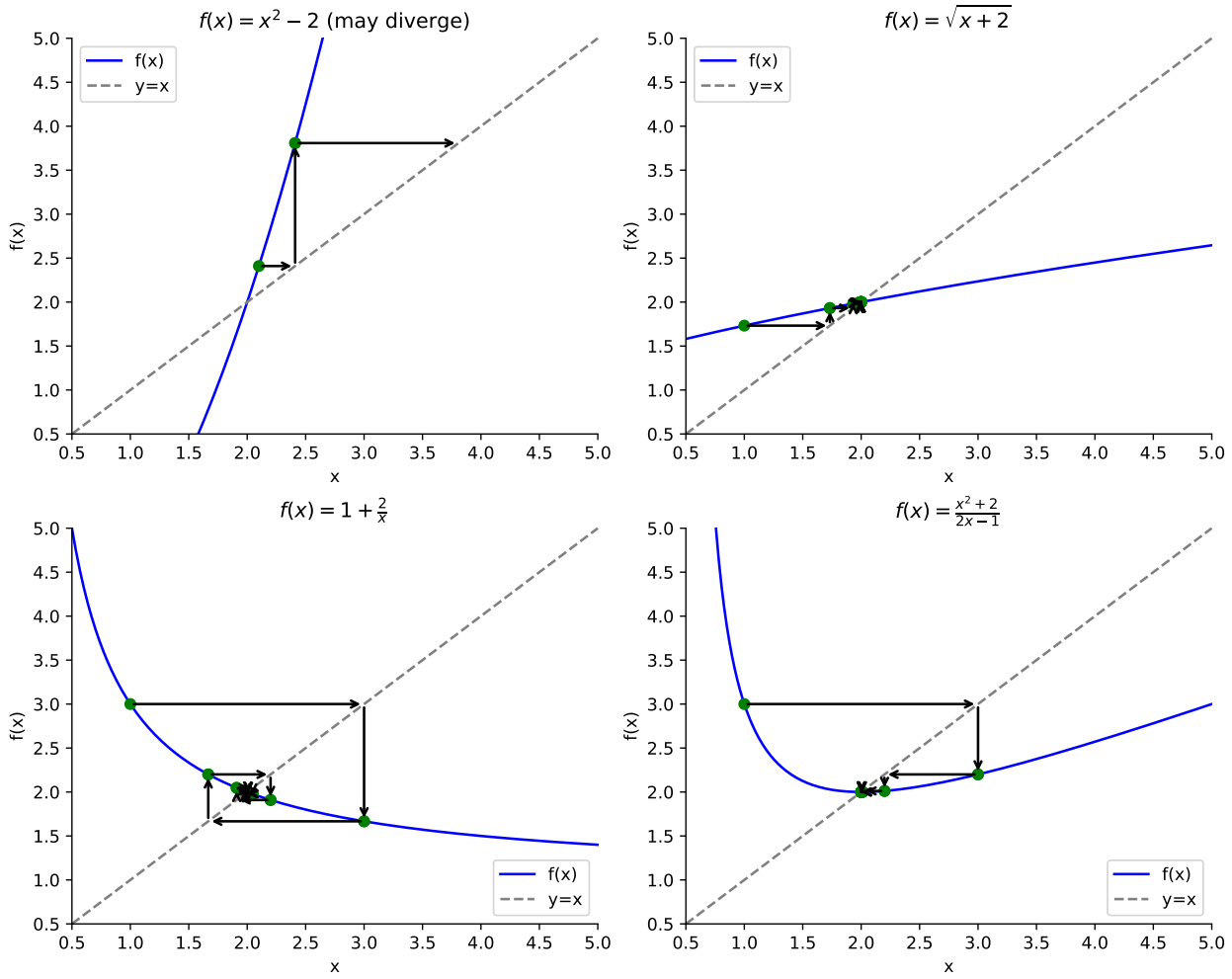
(continued from previous page)

```
    """Plot fixed-point iterations with four functions in subplots."""
    functions = [func1, func2, func3, func4]
    titles = [
        r"$f(x) = x^2 - 2$ (may diverge)",
        r"$f(x) = \sqrt{x + 2}$",
        r"$f(x) = 1 + \frac{2}{x}$",
        r"$f(x) = \frac{x^2 + 2}{2x - 1}$",
    ]
    initial_guesses = [2.1, 1, 1, 1]  # Starting points selected for␣
↪demonstration

    # Generate subplots
    x_vals = np.linspace(0.5, 5, 400)
    plt.close("fixed_point")
    fig, axs = plt.subplots(2, 2, figsize=(10, 8), num="fixed_point")
    axs = axs.flatten()

    for i, (f, title, x0) in enumerate(
        zip(functions, titles, initial_guesses, strict=True)
    ):
        ax = axs[i]
        y_vals = f(x_vals)

        # Run fixed-point iteration
        results = fixed_point_iteration(f, x0, max_iter=10)

        # Plot function and diagonal y=x line
        ax.plot(x_vals, y_vals, label="f(x)", color="blue")
        ax.plot(x_vals, x_vals, label="y=x", color="gray", linestyle="--")

        # Plot iteration trajectory with arrows for each step
        for j in range(0, len(results) - 2, 1):
            x_start, y_start = results[j], f(results[j])
            x_proj = results[j + 1]
            # Horizontal arrow for projection step
            ax.annotate(
                "",
                xy=(x_proj, y_start),
                xytext=(x_start, y_start),
                arrowprops=dict(arrowstyle="->", color="black", lw=1.5),
            )

            # Vertical arrow for function step
            ax.annotate(
                "",
                xy=(x_proj, f(x_proj)),
                xytext=(x_proj, y_start),
                arrowprops=dict(arrowstyle="->", color="black", lw=1.5),
            )

            # Mark the function step point
            ax.plot(x_proj, f(x_proj), "go")  # Mark the next function step␣
↪point
```

(continues on next page)

(continued from previous page)

```
        # Set labels and titles
        ax.set_title(title)
        ax.set_xlim(0.5, 5)
        ax.set_ylim(0.5, 5)
        ax.set_xlabel("x")
        ax.set_ylabel("f(x)")
        ax.legend()


# Execute the plot function
plot_fixed_point_iterations()
```



One can obtain the same result using SciPy. Note that 20 iterations is not sufficient to reach the default relative error tolerance of 10^{-8}.

```
optimize.fixed_point(func3, x0=1, method="iteration", maxiter=30)
```

```
np.float64(1.9999999944120646)
```

The simplest way to characterize the behavior of an iterative scheme $x_{k+1} = g(x_k)$ for a fixed-point problem $x = g(x)$ is to look at the derivative of g in the solution x^*. It is a rule that if $x^* = g(x)$ and $\|g'(x^*)\| < 1$, then the iterative scheme is **locally convergent**. If however $\|g'(x^*)\| > 1$, then the scheme diverges for every initial value different from x^*.

Proof

If x^* is a fixed point, then the error at the k-th iteration is

$$e_{k+1} = x_{k+1} - x^* = g(x_k) - g(x^*)$$

There exist a point θ_k between x_k and x^* for which

$$g(x_k) - g(x^*) = g'(\theta_k)(x_k - x^*)$$

so

$$e_{k+1} = g'(\theta_k)e_k$$

We do not know the value of θ_k, but if $\|g'(x^*)\| < 1$, then by starting the iteration sufficiently close to x^*, there exists a constant C for which $\|g'(\theta_k)\| \leq C < 1$, for $k = 0, 1, \ldots$.

Thus we have

$$\|e_{k+1}\| \leq C\|e_k\| \leq \ldots \leq C^k\|e_{e_0}\|$$

As $C < 1$ implies $C^k \to 0$, also $\|e_k\| \to 0$ and the sequence converges.

The convergence rate of the iterative scheme is linear with $C = \|g'(x^*)\|$. The smaller this constant, the faster the convergence. Ideally, we have $\|g'(x^*)\| = 0$, in which case the Taylor expansion gives

$$g(x_k) - g(x^*) = g''(\xi_k)(x_k - x^*)^2/2$$

with ξ_k between x_k and x^*. This yields

$$\lim_{k\to\infty} \frac{\|e_{k+1}\|}{\|e_k\|^2} = \frac{g''(x^*)}{2}$$

In this case, the *rate of convergence becomes quadratic*. In the next sections we'll see methods to systematically choose g to reach this quadratic convergence.

5.5.3 Newton's method

The bisection method does not make use of the function values (except for their sign), so it is reasonable to assume that better convergence can be achieved by also making use of their magnitude.

We start from the truncated Taylor series

$$f(x + h) \approx f(x) + hf'(x),$$

which is a linear function of h that approximates f near a given x. Its zero is easily determined to be $h = -f(x)/f'(x)$, assuming that $f'(x) \neq 0$. Because the zeros of both functions are not identical, this procedure is repeated in an iterative scheme, called **Newton's method**

This method can be seen as a systematic way of transforming a nonlinear equation $f(x) = 0$ into a fixed-point problem $x = g(x)$, where

$$g(x) = x - f(x)/f'(x)$$

Convergence

To study the convergence of this scheme we determine the derivative

$$g'(x) = f(x)f''(x)/(f'(x))^2$$

- For simple roots ($f(x^*) = 0$ and $f'(x^*) \neq 0$), $g'(x^*) = 0$. Thus the asymptotic convergence rate of Newton's method is quadratic.
- For a multiple root with multiplicity m, it is only linearly convergent, with constant $C = 1 - (1/m)$.

> **Proof**
>
> Generally, you can write a function with a root of multiplicity M at $x = x^*$ as $f(x) = (x - x^*)^M$
>
> As shown earlier, the constant C of linear convergence is given by
>
> $$\|g'(x^*)\| = \|f(x)f''(x)/(f'(x))^2\|$$
>
> filling in
>
> - $f(x) = (x - x^*)^M$
> - $f'(x) = M(x - x^*)^{(M-1)}$
> - $f''(x) = M(M - 1)(x - x^*)^{(M-2)}$
>
> yields $C = 1 - 1/M$

Take note that these convergences are only local and it may not converge at all unless started sufficiently close to the solution.

```
def func_cube(x):
    """Example function for newton and secant methods."""
    return x**3 - 1  # a function with only one real root at x = 1


def func_cube_prime(x):
    """Derivative of func_cube."""
    return 3 * x**2


def newton_method(f, fp, x, niter):
    """Illutrative implementation of the Newton method.

    Parametes
    ---------
    f
        Function to be rooted.
    fp
        The derivative of the function to be rooted.
    x0
        The initial guess of the solution.
    niter
        The number of iterations.

    Returns
    -------
    root
        The approximation of the root.
    """
    for _ in range(niter):
        x = x - f(x) / fp(x)
    return x
```

(continues on next page)

(continued from previous page)

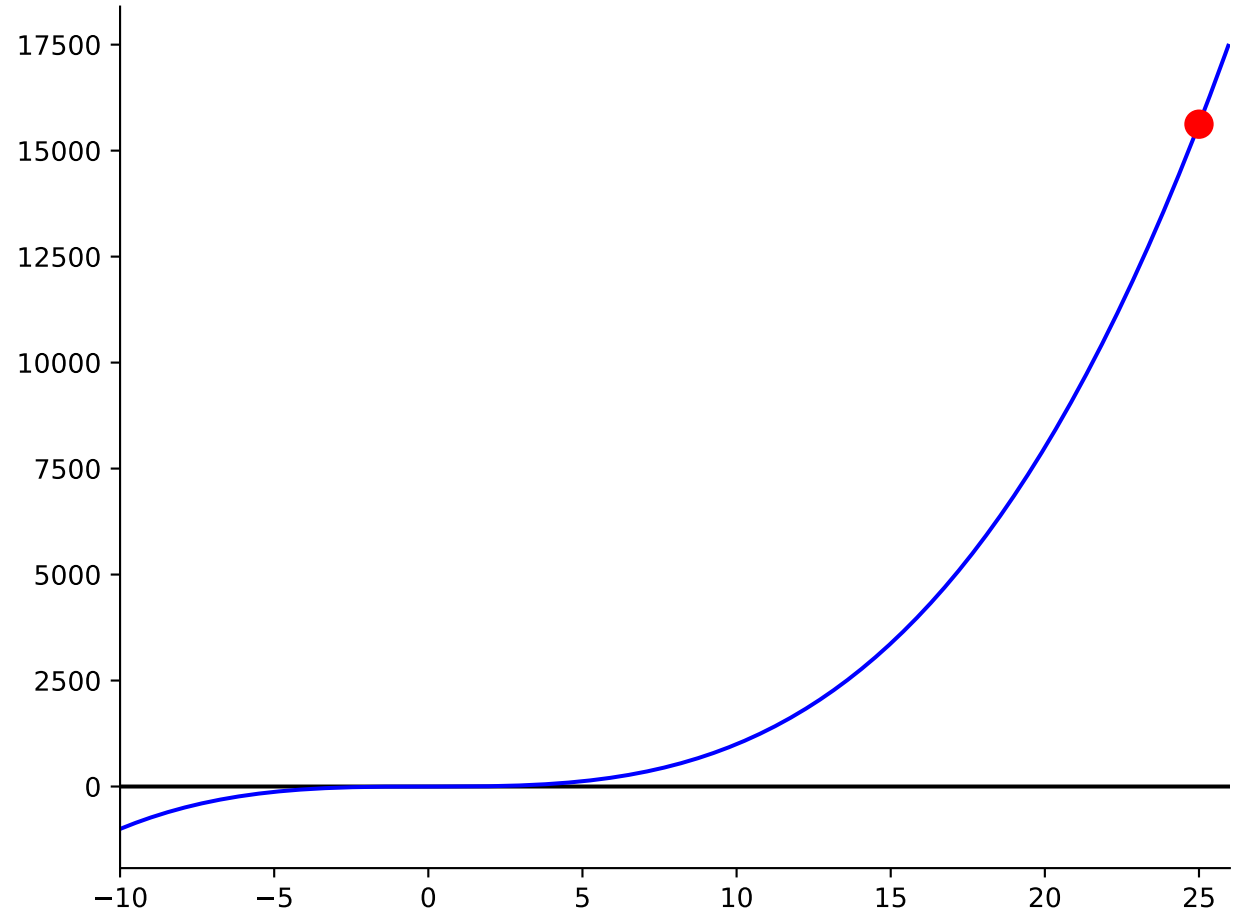
```
# run the implementation for various N values
def newton_method_various_iterations(iterations=20):
    for niter in range(iterations):
        x = newton_method(func_cube, func_cube_prime, 25, niter)
        print(f"{niter:2d} iterations: x = {x:18.15f}")


newton_method_various_iterations()


def plot_and_animate_nm():
    x0 = 25
    x = np.arange(-10, 26, 0.05)
    y = func_cube(x)

    fig, ax = plt.subplots(num="animate_nm", clear=True)

    ax.axhline(0, color="black")
    ax.plot(x, y, color="blue")

    ax.set_xlim(-10, 26)
    # plt.ylim(-2, 20);

    (dots,) = plt.plot([], [], "o", markersize=10, color="r")

    def animate(i):
        x = newton_method(func_cube, func_cube_prime, x0, i)
        y = func_cube(x)
        dots.set_data([x], [y])
        return (dots,)

    return FuncAnimation(fig, animate, frames=20, interval=1000, repeat=False)


plt.close("animate_nm")
nm_anim = plot_and_animate_nm()
```

```
 0 iterations: x = 25.000000000000000
 1 iterations: x = 16.667200000000001
 2 iterations: x = 11.112666589870354
 3 iterations: x =  7.411143637442940
 4 iterations: x =  4.946831301244156
 5 iterations: x =  3.311509021939091
 6 iterations: x =  2.238069410889822
 7 iterations: x =  1.558593758201963
 8 iterations: x =  1.176281072583854
 9 iterations: x =  1.025098324674696
10 iterations: x =  1.000609487781420
11 iterations: x =  1.000000371173706
12 iterations: x =  1.000000000000138
13 iterations: x =  1.000000000000000
14 iterations: x =  1.000000000000000
```

(continues on next page)

(continued from previous page)

```
15 iterations: x =  1.000000000000000
16 iterations: x =  1.000000000000000
17 iterations: x =  1.000000000000000
18 iterations: x =  1.000000000000000
19 iterations: x =  1.000000000000000
```



The Newton method is also implemented in SciPy:

```
optimize.root_scalar(
    func_cube,
    method="newton",
    x0=25,
    fprime=func_cube_prime,
    xtol=1e-16,
    maxiter=500,
)
```

```
      converged: True
           flag: converged
 function_calls: 27
     iterations: 13
           root: 1.0
         method: newton
```

5.5.4 Secant method

One drawback of Newton's method is that both the function and its derivative needs to be explicitly and evaluated at every iteration. In the **Secant method**, the derivative is replaced by a finite difference approximation on successive iterates:

$$f'(x_k) = \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}$$

The secant method can be interpreted geometrically as approximating the function f by the secant line through the previous two estimates, and taking the zero of this function as the best approximate solution.

Convergence

Compared with Newton's method, the secant method has the advantage of requiring only one new function evaluation per iteration, but has the disadvantage of requiring two starting guesses and converging somewhat more slowly (subquadratically but still faster than linear with $r \approx 1.618$).

The lower cost per iteration often more than offsets the larger number of iterations required, such that the total cost of finding a root is often less for the secant method than for Newton's method.

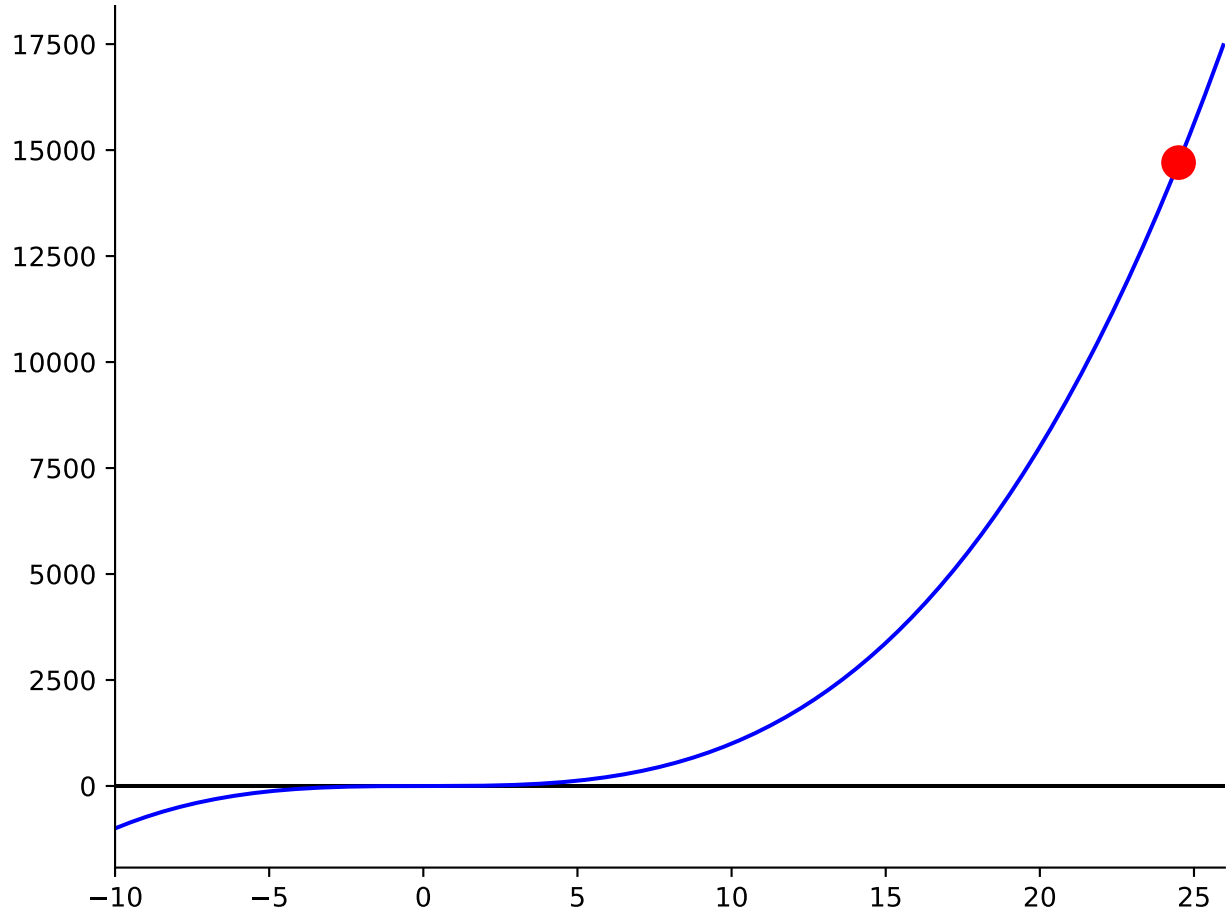
```
def secant_method(f, x0, x1, niter):
    """Illustrative implementation of the secant method.

    Parameters
    ----------
    f
        The function to be rooted.
    x0, x1
        Two different initial guesses.
    niter
        The number of iterations

    Returns
    -------
    root
        The approximate root
    """
    fx0 = f(x0)
    for _ in range(niter):
        temp = x1
        fx1 = f(x1)
        x1 = x1 - fx1 * (x1 - x0) / (fx1 - fx0)
        x0 = temp
        fx0 = fx1
    return (x0 + x1) / 2


def secant_method_various_iterations(iterations=20):
    for niter in range(iterations):
        x = secant_method(func_cube, 25, 24, niter)
        print(f"{niter:2d} iterations: x = {x:18.15f}")


secant_method_various_iterations()
```

(continues on next page)

(continued from previous page)

```
def plot_and_animate_sm(f):
    x0 = 25
    x1 = 24
    x = np.arange(-10, 26, 0.05)
    y = func_cube(x)

    fig, ax = plt.subplots(num="animate_sm", clear=True)
    ax.axhline(0, color="black")
    ax.plot(x, y, color="blue")
    ax.set_xlim(-10, 26)

    (dots,) = plt.plot([], [], "o", markersize=12, color="r")

    def animate(i):
        x = secant_method(func_cube, x0, x1, i)
        y = func_cube(x)
        dots.set_data([x], [y])
        return (dots,)

    return FuncAnimation(fig, animate, frames=20, interval=1000, repeat=False)


plt.close("animate_sm")
sm_anim = plot_and_animate_sm(func_cube)
```

```
 0 iterations: x = 24.500000000000000
 1 iterations: x = 20.162409772348695
 2 iterations: x = 14.562858816022644
 3 iterations: x = 11.161230926417350
 4 iterations: x =  8.376227777455009
 5 iterations: x =  6.340709596703303
 6 iterations: x =  4.787770476664006
 7 iterations: x =  3.625138827537755
 8 iterations: x =  2.753172260249217
 9 iterations: x =  2.107903944561893
10 iterations: x =  1.641652649967808
11 iterations: x =  1.323019096090060
12 iterations: x =  1.129388873032998
13 iterations: x =  1.035808063558681
14 iterations: x =  1.005450630948724
15 iterations: x =  1.000304337274550
16 iterations: x =  1.000003082006963
17 iterations: x =  1.000000001855094
18 iterations: x =  1.000000000000012
19 iterations: x =  1.000000000000000
```



The secant method is also implemented in SciPy:

```
optimize.root_scalar(
    func_cube, method="secant", x0=25, x1=24, xtol=1e-16, maxiter=50
)
```

```
      converged: True
           flag: converged
 function_calls: 20
     iterations: 19
           root: 1.0
         method: secant
```

5.5.5 Inverse Interpolation

The secant method fits a straight line to two values of the function for each iteration. Its convergence rate can be improved (but not made to exceed $r = 2$) by fitting a higher order polynomial instead of a straight line.

This has however the drawbacks that the zeros of the fitted polynomial might be difficult to compute, or might not exist at all.

Instead, we can use **inverse interpolation** where, instead of fitting a polynomial to values $f(x_k)$ as function of the values x_k, we do the opposite: we fit a polynomial p to the values x_k as a function of the values $f(x_k)$. The next approximate solution is than simply $p(0)$.

The most used implementation of this idea is **inverse quadratic interpolation** where a parabola is fitted through the values obtained at the last 3 iterations. Similar to the secant method this only requires one additional function evaluation

per iteration, but requires a little more memory and overhead in fitting the parabola. This algorithm has a converge rate of $r \approx 1.839$.

5.5.6 Root finding in SciPy

The best and safest method used to find the roots of a one-dimensional function is the **Brent method** `optimize.brentq` This is a so-called **safeguarded method**, which combines the safety of (slow) bracket method like the bisection method and the high converge rates of inverse quadratic interpolation.

This method works by defining a suitable starting bracket (at the end of which the function has a different sign), and trying a fast-convergence method. If this method does not converge and the next approximate solution falls outside the defined bracket, the method falls back on the bisection method for one iteration to reduce the size of the bracket and try the fast method (with a higher chance for success) again until the solution is found.

A more detailed explanation can be found on: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.brentq.html#scipy.optimize.brentq

As we'll see below, in multiple dimensions we'll need the `optimize.root` function, which also works in 1 dimension.

```
def func_par(x):
    """A simple quadratic test function with two obvious roots."""
    return x**2 - 1


# find its root in the bracket -2 and 0
print(f"Root in bracket [-2, 0]: {optimize.brentq(func_par, -2, 0)} \n")

# find its root in the bracket 0 and 2
print(f"Root in bracket [0, 2]: {optimize.brentq(func_par, 0, 2)} \n")

# now search for its root using the optimize.root function, with 2 as initial␣
↪guess
optimize.root(func_par, x0=2)
```

```
Root in bracket [-2, 0]: -1.0

Root in bracket [0, 2]: 1.0
```

```
 message: The solution converged.
 success: True
  status: 1
     fun: [ 4.441e-16]
       x: [ 1.000e+00]
  method: hybr
    nfev: 11
    fjac: [[-1.000e+00]]
       r: [-2.000e+00]
     qtf: [-3.824e-10]
```

Example

The force F acting on an object in free fall is given by the sum of the gravitational force $F_g = -mg$ (with m the mass and g the gravitational constant), and the air resistance $F_r = C\rho A v^2/2$ (with C the drag constant of the object, ρ the density of the fluid the object falls through, A the projected surface and v the velocity).

For an object starting at position 0 at time 0, this leads to the following equation of motion:

$$F = m\frac{d^2x}{dt^2} = -mg + \frac{C\rho A}{2}\left(\frac{dx}{dt}\right)^2$$

The analytical solution to this differential equation is given by the nonlinear equation

$$x(t) = -\frac{\log(\cosh(\sqrt{(AgmC\rho/2)}t))}{(AC\rho/2)}$$

Let's now consider 2 skydivers: first a quite big person who jumps in a horizontal position The second skydiver is a 50 kg adolescent who enthusiastically dives down head-first to minimize her air resistance, but who starts 1 second later.

The graph below shows that the first skydiver almost immediately reaches his terminal velocity, whereas the second skydiver needs a bit more time to accelerate, but due to her more streamlined position reaches a higher velocity and eventually overtakes the first.

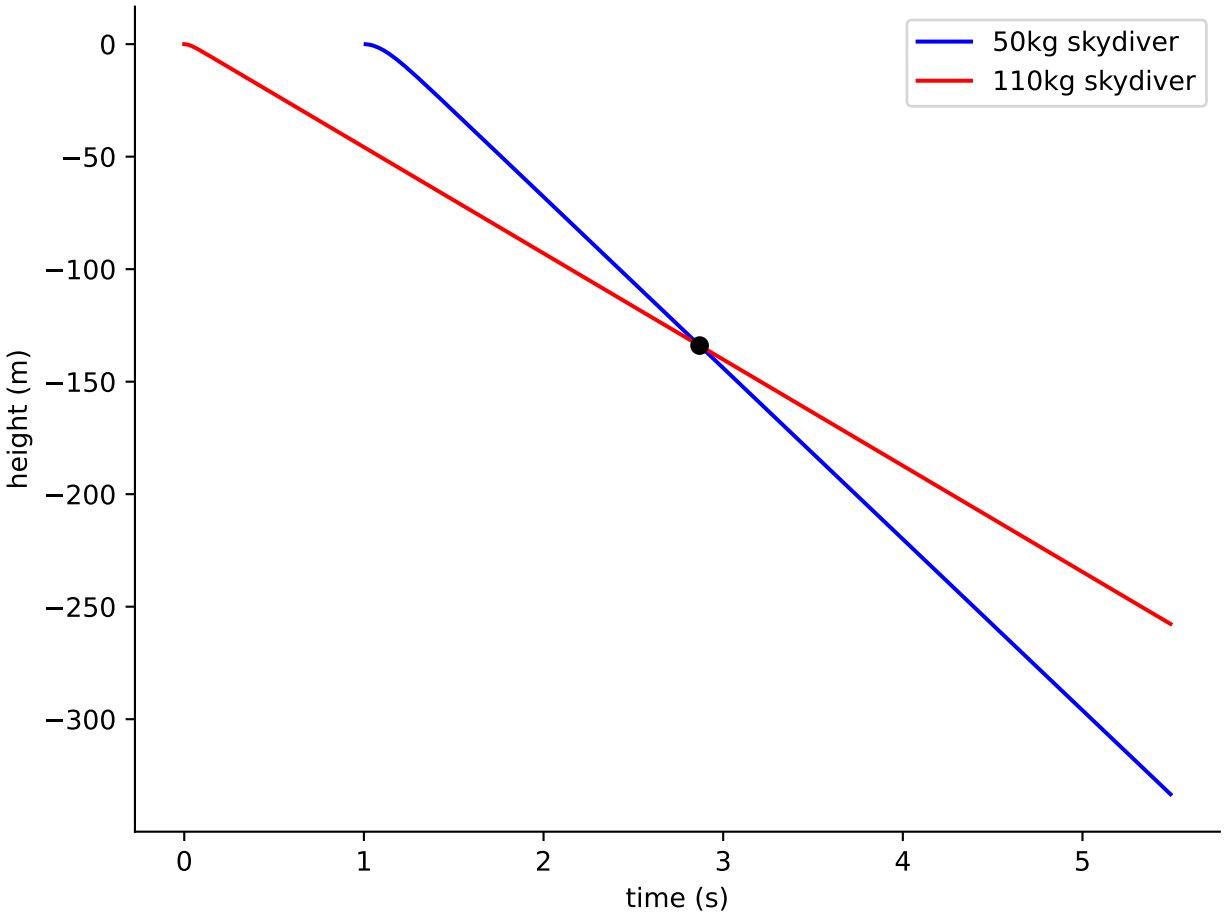
the question we want to answer is when the second one overtakes the first

```
def skydivers():
    # define the freefall equation
    def freefall(x, A, m, C):
        g = 9.8  # m/s²
        r = 1.21  # kg/m³
        return (
            -1.0
            * np.log(np.cosh(np.sqrt(A * g * m * C * r / 2) * x))
            / (A * C * r / 2)
        )

    # Parameters for both skydivers:
    #      Skydiver 1: A = 0.2 m^2,  m =  50 kg, C = 0.7
    #      Skydiver 2: A = 0.8 m^2,  m = 110 kg, C = 1.0

    # skydiver 1
    def skydiver1(x):
        A1, m1, C1 = 0.2, 50.0, 0.70
        return freefall((x - 1), A1, m1, C1)

    # skydiver 2
    def skydiver2(x):
        A2, m2, C2 = 0.8, 110.0, 1.0
        return freefall(x, A2, m2, C2)

    # define x values
    x = np.arange(0, 5.5, 0.01)

    # plot trajectories for both skydivers
    plt.close("skydivers")
    fig, ax = plt.subplots(num="skydivers")

    ax.plot(x[x > 1], skydiver1(x[x > 1]), color="blue", label="50kg skydiver
↪")
    ax.plot(x, skydiver2(x), color="red", label="110kg skydiver")
    ax.set_xlabel("time (s)")
    ax.set_ylabel("height (m)")
    ax.legend()
```

(continues on next page)

(continued from previous page)

```
    # The question we want to answer for this problem is:
    # "At what time does the first skydiver overtake the second one?"

    def fall(x):
        return skydiver1(x) - skydiver2(x)

    x_root = optimize.brentq(fall, 1.01, 5.0)
    y_root = skydiver1(x_root)

    # Plot the intersection point
    ax.plot(x_root, y_root, "o", color="black")


skydivers()
```



5.5.7 Roots of polynomial functions

All methods we saw until now zoom in on a single root of the function under study. Sometimes we're interested in all the roots of e.g. a polynomial function.

For a polynomial $p(x)$ of degree n, we want to find all n zeros (which might be complex).

To this end we can resort to several methods:

- Use one of the methods shown above to find one root x_1 and then deflate the polynomial $p(x)$ to $p(x)/(x - x_1)$ which has a degree that is one lower and repeat the process. Note that it's a good idea to zoom in on each of

the obtained roots using the approximate values used this way to avoid any numerical errors introduced in the deflating process.

- Use a dedicated (complex) routine specifically designed for this purpose. These work by isolating the roots of a polynomial in the complex plane, and then refining in a way similar to the bisection method to zoom in on each of the roots. Their complexity is beyond the scope of this course.
- Form the **companion matrix** of the given polynomial and use an eigenvalue routine to find its eigenvalues, which are also the roots of the polynomial.

The latter method is the one that is used by NumPy in the `numpy.polynomial.polynomial.polyroots` function.

> **Eample:**
>
> Find the roots of the polynomial $x^3 - 6x^2 + 11x - 6$, which equals $(x-1)(x-2)(x-3)$. Note that the `polyroots` function asks an array of polynomial coefficients as input:

```
poly.polyroots([-6, 11, -6, 1])
```

```
array([1., 2., 3.])
```

5.6 Systems of nonlinear equations

Systems of nonlinear equations are more difficult to solve than single nonlinear equations for a number of reasons:

- A much wider range of behavior is possible, so we don't get as far with theoretical analysis of the existence and number of solutions.
- There is no simple way to bracket a desired solution.
- Computational overhead increases rapidly with the dimension of the problem.

Most of the methods we saw to solve a 1-dimensional nonlinear problem do not generalize to more than 1 dimension. One method that does is Newton's method:

For a differentiable vector function $\mathbf{f}$, the truncated Taylor series reads:

$$\mathbf{f}(\mathbf{x} + \mathbf{s}) \approx \mathbf{f}(\mathbf{x}) + \mathbf{J}_{f(\mathbf{x})}\,\mathbf{s}$$

where $\mathbf{J}_{\mathbf{f}(\mathbf{x})}$ is the Jacobian matrix of $\mathbf{f}$ with elements

$$\left[\mathbf{J}_{\mathbf{f}(\mathbf{x})}\right]_{ij} = \frac{\partial f_i(\mathbf{x})}{\partial x_j}$$

If $\mathbf{s}$ satisfies the linear system $\mathbf{J}_{f(\mathbf{x})}\,\mathbf{s} = -\mathbf{f}(\mathbf{x})$, then $\mathbf{x} + \mathbf{s}$ is taken as an approximate zero of $\mathbf{f}$.

Essentially, Newton's method replaces a system of nonlinear equations with a system of linear equations, but as the solutions of both systems are not identical, the process must be repeated until the desired accuracy is reached.

If the Jacobian of the function is not available, there exist more advanced methods which estimate the Jacobian based on function evaluations, similar to how the secant method works in 1 dimension.

The computational cost of Newton's method in n dimensions is substantial:

- Evaluating the Jacobian matrix (or approximating it) requires n^2 function evaluations.
- Solving the system $\mathbf{J}_{\mathbf{f}(\mathbf{x})}\,\mathbf{s} = -\mathbf{f}(\mathbf{x})$, for instance using LU-factorization, costs $\mathcal{O}(n^3)$ operations.

Without going into too much detail in how these methods work, we'll have a look how the solution to such problems can be found using `scipy` using the `optimize.root` function. Its documentation can be found here https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.root.html#scipy.optimize.root

Its use is illustrated with an example:

> **Example**
>
> Solve the nonlinear system
>
> $$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} x_1 + 2x_2 - 2 \\ x_1^2 + 4x_2^2 - 4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

```
def func_2d(x):
    return [x[0] + 2 * x[1] - 2, x[0] ** 2 + 4 * x[1] ** 2 - 4]


# find the roots of this equation with the point [2, 2] as initial guess
optimize.root(func_2d, [2, 2])
```

```
 message: The solution converged.
 success: True
  status: 1
     fun: [ 0.000e+00  0.000e+00]
       x: [ 1.094e-16  1.000e+00]
  method: hybr
    nfev: 15
    fjac: [[-8.776e-01 -4.794e-01]
           [ 4.794e-01 -8.776e-01]]
       r: [-1.139e+00 -6.114e+00 -7.021e+00]
     qtf: [ 2.040e-12  3.734e-12]
```

> We find the solution $\mathbf{x}^* = [0, 1]^\intercal$

CHAPTER 6

Optimization

```
import matplotlib.pyplot as plt
import numpy as np
from numpy.linalg import lstsq, multi_dot
from numpy.polynomial.polynomial import Polynomial as Poly
from scipy import linalg, optimize
```

6.1 Introduction

Optimization problems arise in all areas of science (and are perhaps even more common in engineering, business and industry). Any design does not only have a requirement that it works, but also involves the optimization of some figure of merit, like cost or efficiency. Out of all possible designs, we want the one that optimizes some objective.

Of course, the result will depend on the objective: an engine optimized to deliver maximum power will be very different from one optimized for fuel efficiency.

Next to the objective function, typically there are also some **constraints** that need to be fulfilled. For instance, when designing a bridge, we can optimize for weight or cost, but we still need to make sure it has a certain minimum strength. Among all feasible choices, however, we want to find the one that optimizes the cost/weight.

There is also a certain duality that constraints can become objectives and vice versa. There is an intimate relationship between such dual problems, whose solutions are often identical. For example, the lightest bridge that can support a certain load, typically is the strongest bridge given its weight.

From the previous description, it seems that optimization problems are mainly the concern of people who design certain objects like bridges, engines,… However, also physical systems evolve towards a configuration of minimum energy, which we can study using the optimization techniques found in this notebook.

6.1.1 Concepts and notation

An optimization problem can be expressed mathematically as the problem of determining an argument for which a given function has an extreme value (minimum or maximum) on a given domain.

Formally, given a function $f : \mathbb{R}^n \to \mathbb{R}$, and a set $S \subseteq \mathbb{R}^n$, we seek $\mathbf{x}^* \in S$ such that f attains a minimum on S at $\mathbf{x}^*$, i.e. $f(\mathbf{x}^*) \leq f(\mathbf{x})$ for all $\mathbf{x} \in S$.

Such a point $\mathbf{x}^*$ is called a **minimizer**, or simply a **minimum** of f. A maximum of f is a minimum of $-f$, so it suffices to consider minimization.

The **objective function** f may be linear or nonlinear, and it is usually assumed to be differentiable. The set S is usually defined by a set of equations and inequalities, called **constraints**, which may be linear of nonlinear. Any vector $\mathbf{x} \in S$, i.e. that satisfies the constraints, is called a **feasible point**, and S is called the **feasible set**. If $S = \mathbb{R}^n$, the problem is **unconstrained**

A general **continuous** optimization problem (note that we will not address **discrete** optimization problems) has the form

$$\min_{\mathbf{x}} f(\mathbf{x}) \quad \text{subject to} \quad \mathbf{g}(\mathbf{x}) = \mathbf{0} \quad \text{and} \quad \mathbf{h}(\mathbf{x}) \leq \mathbf{0}$$

where $f : \mathbb{R}^n \to \mathbb{R}$, $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^m$ and $\mathbf{h} : \mathbb{R}^n \to \mathbb{R}^p$.

Optimization problems are classified by the properties of the functions involved. For example if f, $\mathbf{g}$ and $\mathbf{h}$ are all linear, then we have a **linear programming** problem. If any of them are nonlinear, we have a **nonlinear programming** problem. Note that the term programming in optimization has nothing to do with computer programming, but instead refers to planning activities in the sense of management.

What constitutes a solution to an optimization problem? A **global minimum** satisfies $f(\mathbf{x}^*) \leq f(\mathbf{x})$ for *any* feasible point $\mathbf{x}$. Finding such a global minimum, or even verifying that a point is a global minimum is difficult unless the problem has special properties.

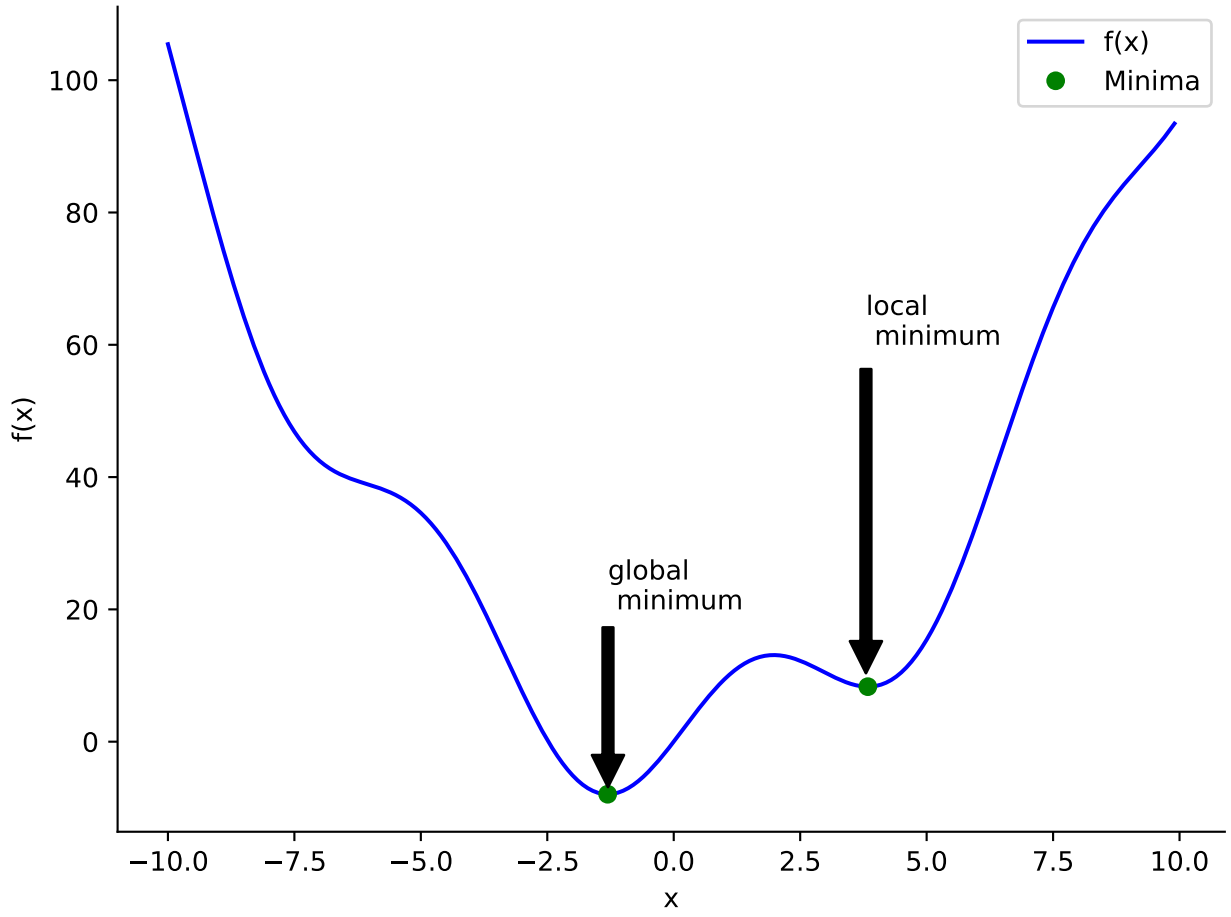
Most optimization methods use local information, such as derivatives, and consequently are designed to find a **local minimum**. Often the best one can do to find a global minimum is use a very large set of starting points, widely scattered throughout the feasible set. The lowest minimum found this way has a good (but not perfect) chance of being the global minimum.

```
def plot_concept():
    def f(x):
        return x**2 + 10 * np.sin(x)

    x = np.arange(-10, 10, 0.1)
    plt.close("concept")
    fig, ax = plt.subplots(num="concept")

    # Plot the function
    ax.plot(x, f(x), "b-", label="f(x)")

    # Plot the minima
    xmins = np.array([-1.30641113, 3.8374671194983834])
    ax.plot(xmins, f(xmins), "go", label="Minima")

    # Decorate the figure
    ax.annotate(
        "global\n minimum",
        xy=(-1.3, f(-1.3)),
        xytext=(-1.3, 20),
        arrowprops=dict(facecolor="black", shrink=0.04),
    )
    ax.annotate(
        "local\n minimum",
        xy=(3.8, f(3.8)),
        xytext=(3.8, 60),
        arrowprops=dict(facecolor="black", shrink=0.04),
    )
    ax.legend(loc="best")
    ax.set_xlabel("x")
```

(continues on next page)

(continued from previous page)

```
    ax.set_ylabel("f(x)")


plot_concept()
```



6.2 Optimality conditions

6.2.1 Unconstrained optimality conditions

The **first-order necessary condition** for a minimum is that the **gradient** of the objective function f is zero.

$$\nabla f(\mathbf{x}^*) = 0$$

Such an $\mathbf{x}^*$ is called a **critical point** and the gradient is defined by

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \frac{\partial f(\mathbf{x})}{\partial x_2} \\ \vdots \\ \frac{\partial f(\mathbf{x})}{\partial x_n} \end{bmatrix}$$

To understand the first-order necessary condition one should remember that the gradient always points *uphill* from $f(\mathbf{x})$. Similarly, the negative gradient, $-\nabla f(\mathbf{x}^*)$ always points *downhill* from $f(\mathbf{x})$. Since there is no *downhill* direction at a minimum, the gradient must be zero. An analogous reasoning applies for a maximum.

Equilibrium

Physically, the same condition corresponds to an equilibrium: a minimum of the potential energy must occur at a critical point of V, i.e. when $\nabla V = 0$. We recall that force is defined by the negative gradient of the potential energy:

$$\mathbf{F}(x) = -\nabla V(x)$$

Thus the negative gradient of the potential energy is zero and the system is in equilibrium when there are no net forces.

In general, it is necessary but not sufficient that $\mathbf{x}$ is a critical point of f for it to be a **minimum** of f. A critical point can either be a minimum, a maximum or a saddle point. To classify the critical points, we need another criterion.

Consider the **Hessian matrix** of f. This is a matrix-valued function $\mathbf{H}_f$ (only defined if f is twice differentiable).

$$\mathbf{H}_f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f(\mathbf{x})}{\partial x_1^2} & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_2} & \frac{\partial^2 f(\mathbf{x})}{\partial x_2^2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_n^2} \end{bmatrix}$$

We can then classify critical points as follows:

At a critical point $\mathbf{x}^*$, where $\nabla f(\mathbf{x}) = \mathbf{0}$, if $\mathbf{H}_f(\mathbf{x}^*)$ is…

- Positive definite, then $\mathbf{x}^*$ is a minimum of f
- Negative definite, then $\mathbf{x}^*$ is a maximum of f
- Indefinite, then $\mathbf{x}^*$ is a saddle point of f
- Singular, then various pathological situations can occur

This is called the **second-order sufficient condition**.

Example

Consider the function

$$f(\mathbf{x}) = 2x_1^3 + 3x_1^2 + 12x_1x_2 + 3x_2^2 - 6x_2 + 6$$

It's gradient is given by

$$\nabla f(\mathbf{x}) = \begin{bmatrix} 6x_1^2 + 6x_1 + 12x_2 \\ 12x_1 + 6x_2 - 6 \end{bmatrix}$$

Solving the nonlinear system $\nabla f(\mathbf{x}) = 0$ yields two critical points, $[1 \quad -1]^\intercal$ and $[2 \quad -3]^\intercal$.

The Hessian matrix is given by

$$\mathbf{H}_f(\mathbf{x}) = \begin{bmatrix} 12x_1 + 6 & 12 \\ 12 & 6 \end{bmatrix}$$

Evaluating $\mathbf{H}_f(\mathbf{x})$ at each of the critical points gives

$$\mathbf{H}_f(1, -1) = \begin{bmatrix} 18 & 12 \\ 12 & 6 \end{bmatrix}$$

and

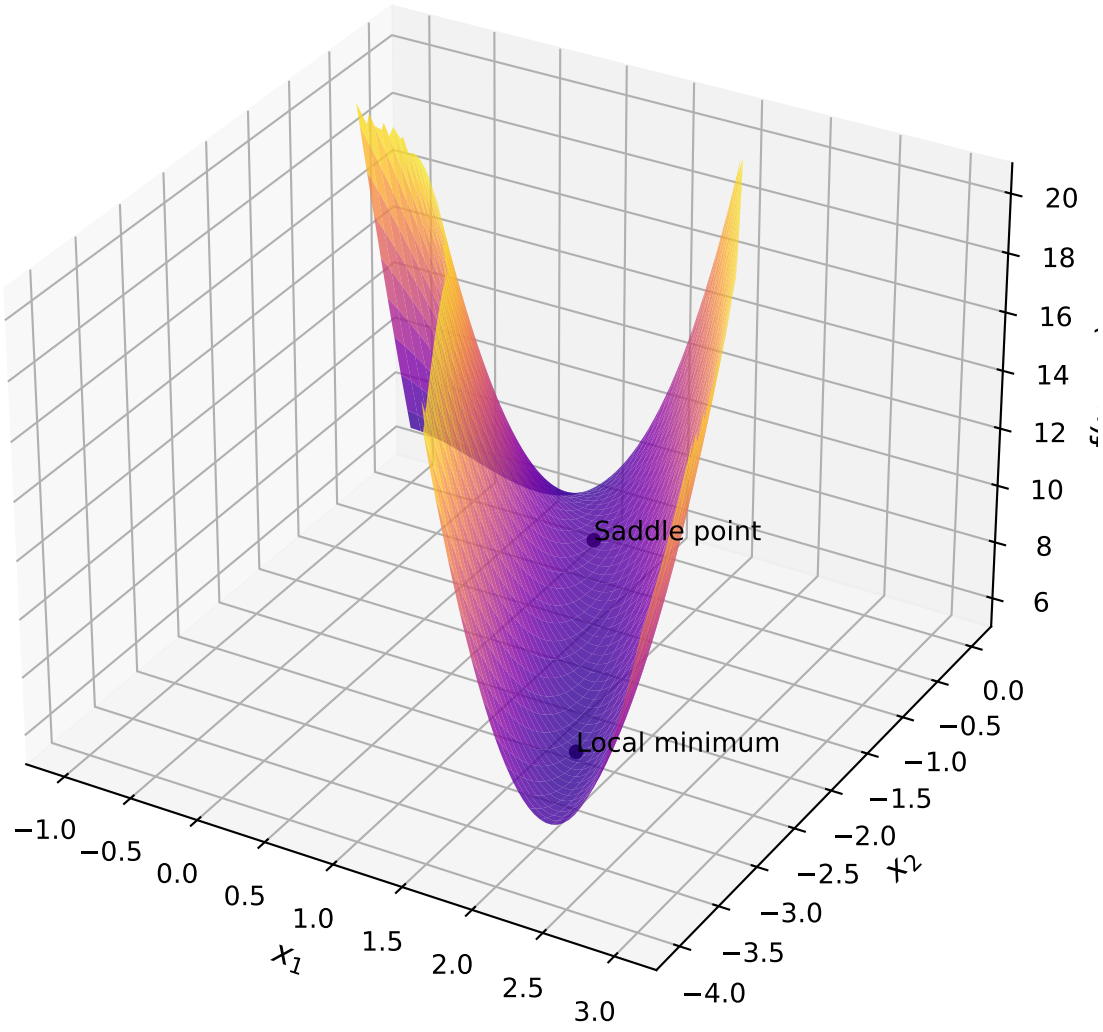
$$\mathbf{H}_f(2,-3) = \begin{bmatrix} 30 & 12 \\ 12 & 6 \end{bmatrix}$$

The Hessian at the first point is not positive definite (its eigenvalues are approximately 25.4 and -1.4), whereas it is positive definite at the second point (with eigenvalues of approximately 35 and 1).

We can therefore conclude that $[1 \quad -1]^\intercal$ is a saddle point and $[2 \quad -3]^\intercal$ is a local minimum of f.

The following cells give a visual representation of the example above. First the function is defined, followed by two cell for the plots. The plots show respectively a 3D and 2D graph of the function $f(\mathbf{x})$. The two critical points are explicitly marked in both graphs.

```
def func_surface(x1, x2):
    """Function from the example above."""
    return 2 * x1**3 + 3 * x1**2 + 12 * x1 * x2 + 3 * x2**2 - 6 * x2 + 6


def plot_example_3d():
    # Create a meshgrid for 3D plot
    x1 = np.linspace(-1, 3, 100)
    x2 = np.linspace(-4, 0, 100)
    x1, x2 = np.meshgrid(x1, x2)
    # Compute the function and truncate high values with nan
    # to show only the surface near the stationary points.
    z = func_surface(x1, x2)
    z[z > 20] = np.nan

    # Create 3D plot
    plt.close("example3d")
    fig = plt.figure(figsize=(6, 6), num="example3d")
    ax = fig.add_subplot(projection="3d")
    ax.plot_surface(x1, x2, z, cmap="plasma", alpha=0.8)

    # Critical points with text
    critical_points = {"Saddle point": [1, -1], "Local minimum": [2, -3]}
    for label, point in critical_points.items():
        ax.scatter(*point, func_surface(*point), c="k")
        ax.text(*point, func_surface(*point), label, color="black", zorder=10)

    # Add labels, legend and title
    ax.set_xlabel("$x_1$", fontsize=12)
    ax.set_ylabel("$x_2$", fontsize=12)
    ax.set_zlabel("f($x_1$, $x_2$)", fontsize=12)


plot_example_3d()
```

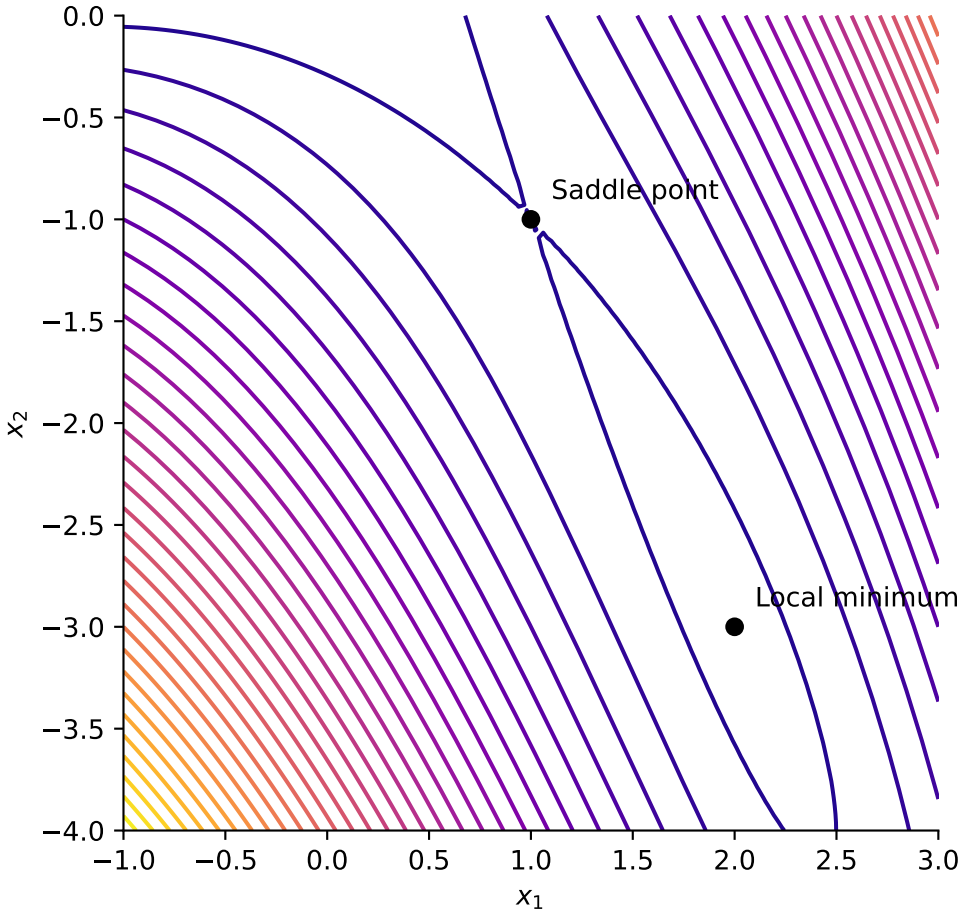



```python
def plot_example_2d():
    # Create a meshgrid for 2D plot
    x1 = np.linspace(-1, 3, 100)
    x2 = np.linspace(-4, 0, 100)
    x1, x2 = np.meshgrid(x1, x2)

    plt.close("example2d")
    fig, ax = plt.subplots(num="example2d")
    ax.contour(x1, x2, func_surface(x1, x2), levels=30, cmap="plasma")
    ax.set_aspect("equal")

    # Critical points with text
    critical_points = {"Saddle point": [1, -1], "Local minimum": [2, -3]}
    for label, point in critical_points.items():
        plt.plot(point[0], point[1], "ko")
        plt.text(point[0] + 0.1, point[1] + 0.1, label, color="black",␣
↪zorder=10)
```

(continues on next page)

(continued from previous page)

```
    # Add labels, legend and title
    ax.set_xlabel("$x_1$")
    ax.set_ylabel("$x_2$")


plot_example_2d()
```



6.3 Optimization in one dimension

The techniques used here are similar to the ones used to find the solution to one-dimensional nonlinear equations, where we used a sign change to bracket the solution.

A function $f : \mathbb{R} \to \mathbb{R}$ is **unimodal** on an interval $[a, b]$ if there is a unique $x^* \in [a, b]$ such that $f(x^*)$ is the minimum value of f on $[a, b]$, and for any $x_1, x_2 \in [a, b]$ with $x_1 < x_2$,

$x_2 < x^*$ implies $f(x_1) > f(x_2)$ and $x_1 > x^*$ implies $f(x_1) < f(x_2)$.

Thus, $f(x)$ is strictly decreasing for $x \leq x^*$ and strictly increasing for $x \geq x^*$. This property will allow us to refine an interval containing a solution by computing sample values of the function within the interval and discarding portions of the interval according to the function values obtained, analogous to bisection for solving nonlinear equations.

6.3.1 Demo function

The same univariate function is used to demonstrate all the implementations below. You can change the definition here and rerun the examples to test the algorithms for a different case.

```
def func_single(x):
    """The demo function."""
    return 0.5 - x * np.exp(-x * x)


def func_single_p(x):
    """First derivative of the demo function."""
    return (2 * x**2 - 1) * np.exp(-x * x)


def func_single_pp(x):
    """Second derivative of the demo function."""
    return 2 * x * (3 - 2 * x**2) * np.exp(-x * x)
```

6.3.2 Golden section search

Suppose f is unimodal on $[a, b]$, and let $x_1, x_2 \in [a, b]$ with $x_1 < x_2$. By comparing the function values $f(x_1)$ and $f(x_2)$ we can exclude a subinterval, either $(x_2, b]$ or $[a, x_1)$ because we know that the minimum lies within the remaining subinterval.

In particular, if $f(x_1) < f(x_2)$, than the minimum cannot lie in the interval $(x2, b]$ and if on the other hand $f(x_1) > f(x_2)$ than the minimum cannot lie in the interval $[a, x_1)$.

This means that we are left with a shorter interval of which we already know one function value. Hence, we only need to calculate one more to repeat this process until our bracket reaches a certain tolerance.

To make consistent progress in reducing the length of the interval containing the minimum, each pair of points in the new interval should have the same relative position as the old pair in the old interval.

To accomplish this objective, we choose the relative positions of the two points to be τ and $1 - \tau$, where $\frac{\tau}{1} = \frac{1-\tau}{\tau} \rightarrow \tau^2 = 1 - \tau$, so that $\tau = (\sqrt{(5)} - 1)/2 \approx 0.618$ (the "golden ratio") and $1 - \tau \approx 0.382$. The complete procedure converges linearly to a local minimum *if* the function is unimodal within the initial bracket.

In the following cells, an algorithm for golden section search is shown, together with an example and the corresponding `scipy` command to solve the same example problem.

```
def my_golden(f, a, b, tol):
    """Illustrative implementation of the Golden Section
    method with visualization."""
    brackets = [[a, b]]
    i = 0
    print("  i           x1           x2        x2-x1")
    print("---  -----------  -----------  -----------")
    print(f"{0:3d}  {a:11.9f}  {b:11.9f}  {b-a:11.9f}")
    t = (np.sqrt(5) - 1) / 2
    x1 = a + (1 - t) * (b - a)
    f1 = f(x1)
    x2 = a + t * (b - a)
    f2 = f(x2)
    points = [[x1, x2]]

    while (b - a) > tol:
        if f1 > f2:
```

(continues on next page)

(continued from previous page)

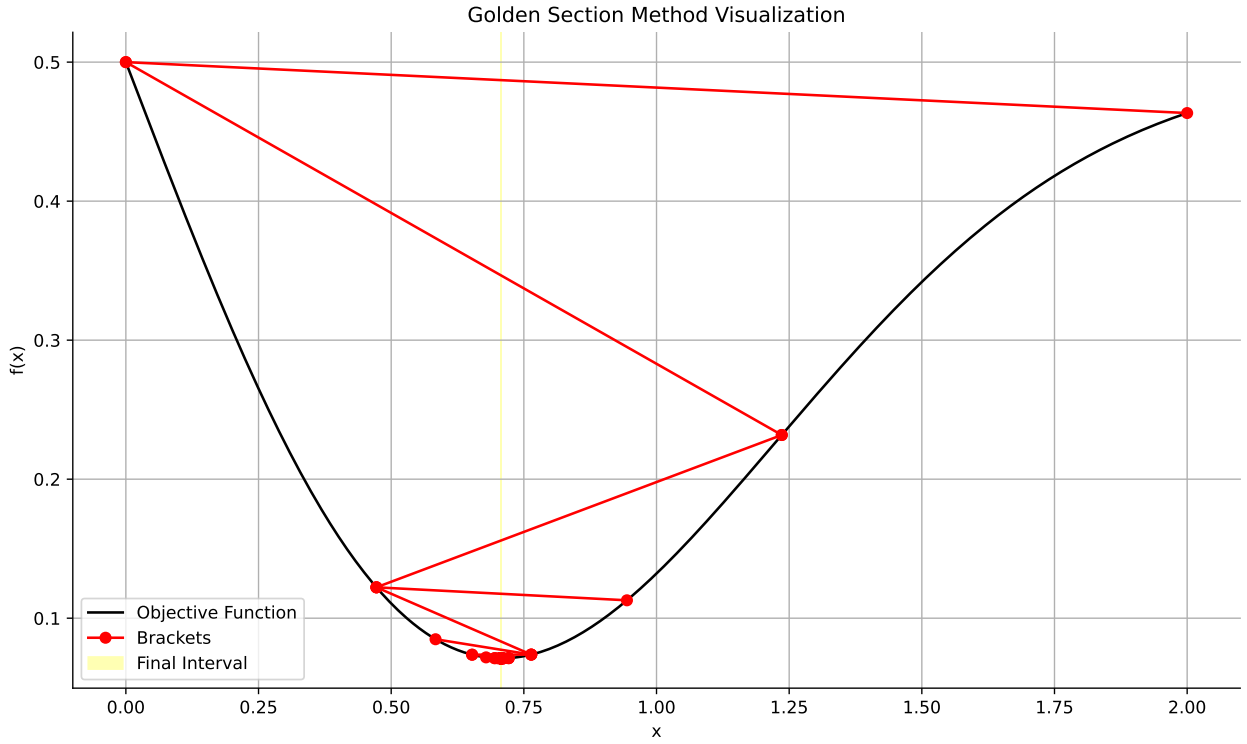
```
            a = x1
            x1 = x2
            f1 = f2
            x2 = a + t * (b - a)
            f2 = f(x2)
        else:
            b = x2
            x2 = x1
            f2 = f1
            x1 = a + (1 - t) * (b - a)
            f1 = f(x1)
        brackets.append([a, b])
        points.append([x1, x2])
        i = i + 1
        print(f"{i:3d}  {a:11.9f}  {b:11.9f}  {b-a:11.9f}")

    return np.array(brackets), np.array(points)


def plot_golden():
    # Perform the Golden Section method and collect data
    brackets, points = my_golden(func_single, a=0, b=2, tol=1e-6)

    # Generate the static figure
    t = np.linspace(0, 2, 500)
    plt.close("golden")
    fig, ax = plt.subplots(figsize=(10, 6), num="golden")
    ax.plot(t, func_single(t), label="Objective Function", color="black")

    # Plot the brackets and points for each iteration
    for i, (bracket, _point) in enumerate(zip(brackets, points,␣
 ↪strict=False)):
        ax.plot(
            bracket,
            [func_single(bracket[0]), func_single(bracket[1])],
            "r-o",
            label="Brackets" if i == 0 else "",
        )

    # Highlight the final region
    ax.axvspan(
        brackets[-1, 0],
        brackets[-1, 1],
        color="yellow",
        alpha=0.3,
        label="Final Interval",
    )

    ax.set_xlabel("x")
    ax.set_ylabel("f(x)")
    ax.set_title("Golden Section Method Visualization")
    ax.legend()
    ax.grid(True)
```

(continues on next page)

(continued from previous page)

```
plot_golden()
```

```
  i           x1           x2        x2-x1
---  -----------  -----------  -----------
  0  0.000000000  2.000000000  2.000000000
  1  0.000000000  1.236067977  1.236067977
  2  0.472135955  1.236067977  0.763932023
  3  0.472135955  0.944271910  0.472135955
  4  0.472135955  0.763932023  0.291796068
  5  0.583592135  0.763932023  0.180339887
  6  0.652475842  0.763932023  0.111456180
  7  0.652475842  0.721359550  0.068883707
  8  0.678787077  0.721359550  0.042572473
  9  0.695048315  0.721359550  0.026311235
 10  0.695048315  0.711309553  0.016261238
 11  0.701259555  0.711309553  0.010049997
 12  0.705098312  0.711309553  0.006211240
 13  0.705098312  0.708937070  0.003838757
 14  0.706564587  0.708937070  0.002372483
 15  0.706564587  0.708030862  0.001466275
 16  0.706564587  0.707470795  0.000906208
 17  0.706910728  0.707470795  0.000560067
 18  0.706910728  0.707256868  0.000346141
 19  0.707042942  0.707256868  0.000213927
 20  0.707042942  0.707175156  0.000132214
 21  0.707042942  0.707124655  0.000081713
 22  0.707074153  0.707124655  0.000050501
 23  0.707093443  0.707124655  0.000031211
 24  0.707093443  0.707112733  0.000019290
 25  0.707100811  0.707112733  0.000011922
 26  0.707105365  0.707112733  0.000007368
 27  0.707105365  0.707109918  0.000004554
 28  0.707105365  0.707108179  0.000002814
 29  0.707106440  0.707108179  0.000001739
 30  0.707106440  0.707107515  0.000001075
 31  0.707106440  0.707107104  0.000000664
```



6.3.3 Successive parabolic interpolation

The golden section search for optimization is analogous to the bisection method to solve a nonlinear equation. Similarly, we do not make use of the function values, other than to compare them. We can do better by making better use of the function values.

Fitting a straight line to two points, as in the secant method, is useless as this line does not have any minimum. Instead, we must use a polynomial with degree of at least two.

The simplest such approach is **successive parabolic interpolation**, where the function is evaluated at three points, and a parabola is fitted to the resulting function values. The minimum of the parabola is used as a new approximate value of the minimum.

A straightforward implementation in python is shown below, together with an animation of an example problem.

This algorithm is not guaranteed to converge, but if it is started reasonable close to a minimum it converges superlinearly with convergence rate $r \approx 1.324$

```
def my_parabolic_iteration(f, u, v, w, tol):
    """Illustrative implementation of Parabolic Iteration."""
    i = 0
    brackets = []
    brackets.append([v])
    par_points = []
    par_points.append([u, v, w])
    print("  i            v          f(v)")
    print("---  -----------  -----------")
    print(f"{0:3d}  {v:11.9f}  {f(v):11.9f}")
    p = 1.0
    q = 1.0
    while (np.abs(p / q)) >= tol:
        i = i + 1
```

(continues on next page)

(continued from previous page)

```
        p = (v - u) ** 2.0 * (f(v) - f(w)) - (v - w) ** 2.0 * (f(v) - f(u))
        q = 2.0 * ((v - u) * (f(v) - f(w)) - (v - w) * (f(v) - f(u)))
        u = w
        w = v
        v = v - p / q
        print(f"{i:3d}  {v:11.9f}  {f(v):11.9f}")
        brackets.append([v])
        par_points.append([u, w, v])
    return np.array(brackets), np.array(par_points)


# find the zero crossing
pi_results, pi_points = my_parabolic_iteration(
    func_single, u=0.5, v=1.75, w=1.2, tol=1e-8
)

# get coefficients of parabola
pi_coeffs = np.array(
    [Poly.fit(points, func_single(points), 2).convert().coef for points in pi_
 ↪points]
)
```

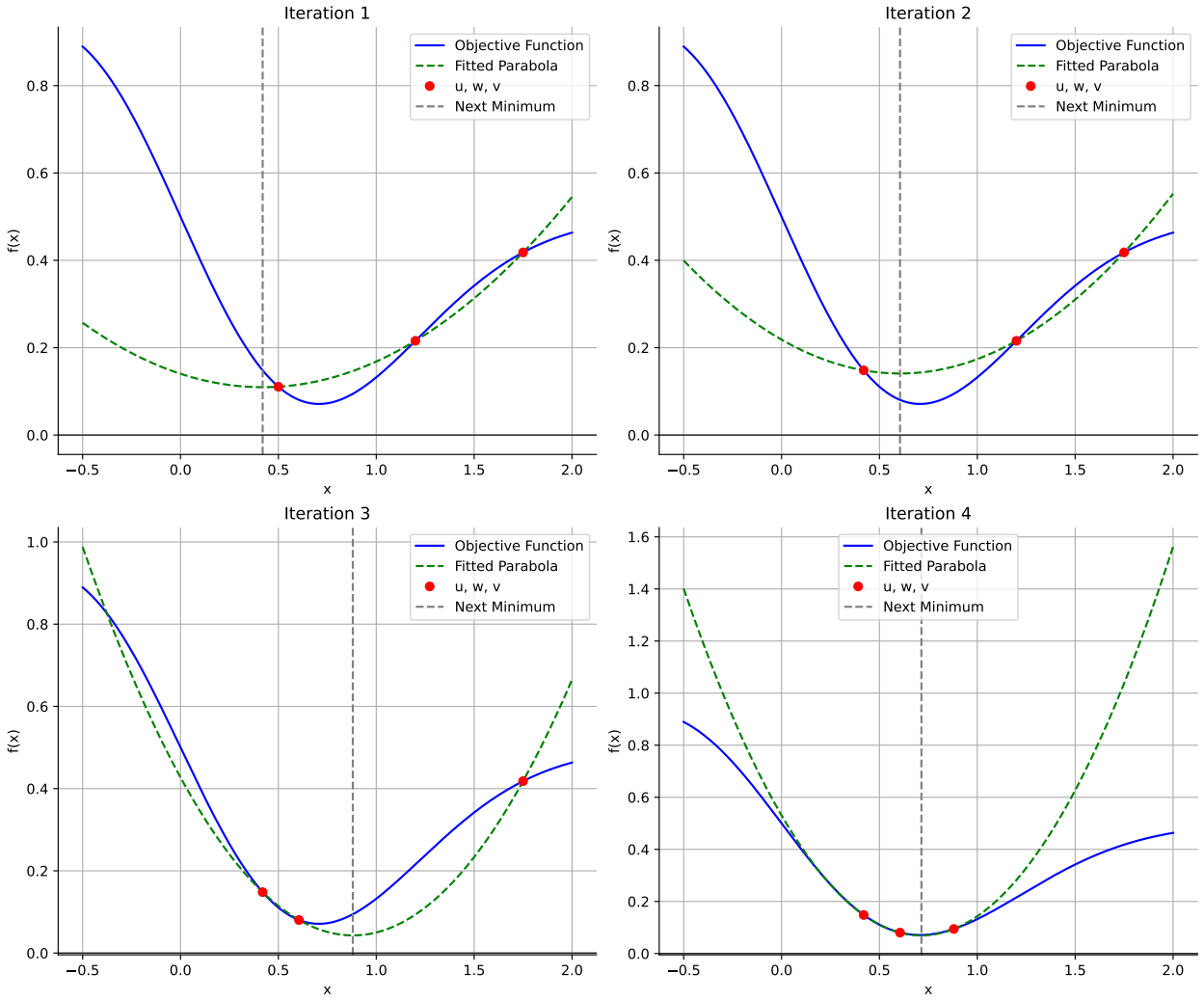
```
  i            v          f(v)
---  -----------  -----------
  0  1.750000000  0.418151411
  1  0.419584192  0.148147059
  2  0.605337612  0.080378668
  3  0.880216960  0.094392264
  4  0.714539112  0.071165273
  5  0.711575541  0.071135151
  6  0.706466178  0.071118410
  7  0.707100640  0.071118058
  8  0.707107465  0.071118058
  9  0.707106780  0.071118058
 10  0.707106781  0.071118058
```

```
def plot_parabolic_iteration():
    # Generate figure
    x = np.linspace(-0.5, 2, 500)
    plt.close("parabolic")
    fig, axs = plt.subplots(2, 2, figsize=(12, 10), num="parabolic_iteration")
    axs = axs.flatten()

    for i, ax in enumerate(axs[:4]):  # Only plot the first 4 iterations
        if i >= len(pi_points):
            break
        points = pi_points[i]
        coeffs = pi_coeffs[i]
        parabola = coeffs[2] * x**2 + coeffs[1] * x + coeffs[0]

        # Plot the function and the fitted parabola
        ax.plot(x, func_single(x), label="Objective Function", color="blue")
        ax.plot(x, parabola, "--", label="Fitted Parabola", color="green")
```

(continues on next page)

(continued from previous page)

```
        # Highlight the points u, w, and v
        ax.plot(points, func_single(points), "ro", label="u, w, v")
        ax.axvline(
            pi_results[i + 1][0], color="grey", linestyle="--", label="Next␣
↪Minimum"
        )

        # Labels and titles
        ax.set_title(f"Iteration {i + 1}")
        ax.set_xlabel("x")
        ax.set_ylabel("f(x)")
        ax.axhline(0, color="black", linewidth=0.8)
        ax.legend()
        ax.grid()


plot_parabolic_iteration()
```



6.3.4 Newton's method

As we saw in the previous method, a quadratic approximation to the objective function is useful because its minimum is easy to compute. Instead of fitting this function, we can also obtain a local quadratic approximation based on a truncated Taylor expansion.

$$f(x+h) \approx f(x) + f'(x)h + \frac{1}{2}f''(x)h^2$$

The minimum of this function is given by $-f'(x)/f''(x)$, which we can use to find the minimum of the objective function in an iterative way.

This method is equivalent to Newton's method for solving nonlinear equations, and also has a quadratic convergence rate. However, unless it is started sufficiently close to the desired minimum it might not converge at all, or converge to a maximum or inflection point instead.

A straightforward implementation in python is shown below, together with an animation of an example problem.

```
def my_newton_method(f, fp, fpp, x, tol):
    """Illustrative implementation of the 1D Newton method."""
    i = 0
    brackets = []
    brackets.append([x])
    print("  i            x         f(x)")
    print("---  -----------  -----------")
    print(f"{0:3d}  {x:11.9f}  {f(x):11.9f}")
    diff = 1.0
    while (np.abs(diff)) >= tol:
        i = i + 1
        diff = fp(x) / fpp(x)
        x = x - diff
        print(f"{i:3d}  {x:11.9f}  {f(x):11.9f}")
        brackets.append([x])
    return np.array(brackets)


results = my_newton_method(
    func_single, func_single_p, func_single_pp, x=1.0, tol=1e-8
)
```

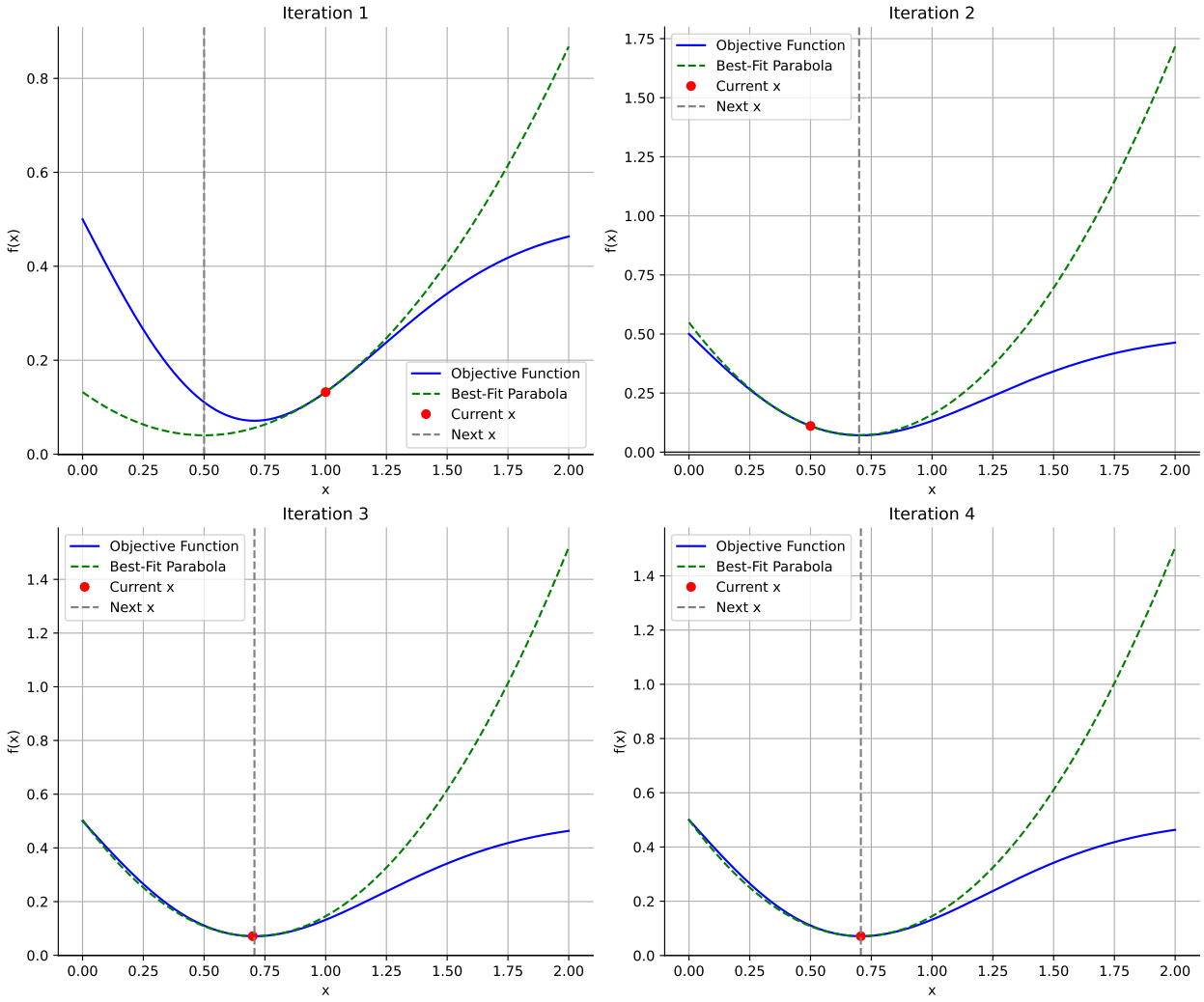
```
  i            x         f(x)
---  -----------  -----------
  0  1.000000000  0.132120559
  1  0.500000000  0.110599608
  2  0.700000000  0.071161524
  3  0.707072136  0.071118059
  4  0.707106780  0.071118058
  5  0.707106781  0.071118058
```

```
def plot_newton_method():
    x_range = np.linspace(0, 2.0, 500)
    plt.close("newton_method")
    fig, axs = plt.subplots(2, 2, figsize=(12, 10), num="newton_method")
    axs = axs.flatten()

    # Only plot the first 4 iterations
    for i, ax in enumerate(axs[:4]):
```

(continues on next page)

(continued from previous page)

```
        if i >= len(results) - 1:
            break

        x_current = results[i][0]
        x_next = results[i + 1][0] if i + 1 < len(results) else x_current

        # Plot the function
        ax.plot(
            x_range, func_single(x_range), label="Objective Function", color=
↪"blue"
        )

        # Plot the best-fit parabola
        f_val = func_single(x_current)
        f_prime = func_single_p(x_current)
        f_double_prime = func_single_pp(x_current)
        parabola = (
            f_val
            + f_prime * (x_range - x_current)
            + 0.5 * f_double_prime * (x_range - x_current) ** 2
        )
        ax.plot(x_range, parabola, "--", label="Best-Fit Parabola", color=
↪"green")

        # Highlight the current point
        ax.plot(x_current, func_single(x_current), "ro", label="Current x")
        ax.axvline(x_next, color="grey", linestyle="--", label="Next x")

        # Labels and titles
        ax.set_title(f"Iteration {i + 1}")
        ax.set_xlabel("x")
        ax.set_ylabel("f(x)")
        ax.axhline(0, color="black", linewidth=0.8)
        ax.legend()
        ax.grid()


plot_newton_method()
```



6.4 Multidimensional unconstrained optimization

We next consider multidimensional unconstrained optimization, which has a number of features in common with both one-dimensional optimization and with solving of linear equations in n dimensions.

> Below, several optimization methods are discussed and as a two-dimensional example, the function
>
> $$f(\mathbf{x}) = 0.5x_1^2 + x_1 + 2.5x_2^2 + 1$$
>
> will be optimized using each method. To illustrate the effectiveness of these methods, they will be compared to each other at the end of this section.
>
> You easily replace this test function by the more challenging Himmelblau function in the following cell.

```
def get_demo_func_quadratic():
    def func(x):
        """Example function to be minimized."""
        return 0.5 * x[0] ** 2 + x[0] + 2.5 * x[1] ** 2 + 1
```

(continues on next page)

(continued from previous page)

```
    def func_p(x):
        """Gradient of the example function."""
        g0 = x[0] + 1
        g1 = 5 * x[1]
        return np.array([g0, g1])

    def func_pp(x):
        """The Hessian of the example function."""
        # Easy: just a constant matrix
        return np.array([[1, 0], [0, 5]])

    return func, func_p, func_pp


def get_demo_func_himmelblau():
    def func(x):
        """Himmelblau function."""
        return (x[0] ** 2 + x[1] - 11) ** 2 + (x[0] + x[1] ** 2 - 7) ** 2

    def func_p(x):
        """Gradient of the Himmelblau function."""
        g0 = 4 * x[0] * (x[0] ** 2 + x[1] - 11) + 2 * (x[0] + x[1] ** 2 - 7)
        g1 = 2 * (x[0] ** 2 + x[1] - 11) + 4 * x[1] * (x[0] + x[1] ** 2 - 7)
        return np.array([g0, g1])

    def func_pp(x):
        """Hessian matrix of the Himmelblau function."""
        h00 = 12 * x[0] ** 2 + 4 * x[1] - 42
        h01 = 4 * x[0] + 4 * x[1]
        h11 = 12 * x[1] ** 2 + 4 * x[0] - 26
        return np.array([[h00, h01], [h01, h11]])

    return func, func_p, func_pp


func_multi, func_multi_p, func_multi_pp = get_demo_func_quadratic()
# func_multi, func_multi_p, func_multi_pp = get_demo_func_himmelblau()
```

To illustrate the result of a single case, the following `plot_opt_traj` function is used.

```
def plot_opt_trajectory(results, num):
    """Plot the optimization trajectory."""
    plt.close(num)
    fig, ax = plt.subplots(figsize=(8, 4), num=num)
    x = np.linspace(-8, 8, 100)
    y = np.linspace(-4.3, 4.3, 100)

    xx, yy = np.meshgrid(x, y, sparse=False)
    zz = func_multi(np.array([xx, yy]))
    clevels = np.array([func_multi(results_sd[i, 0]) for i in range(10)])

    ax.contour(x, y, zz, np.flip(clevels))
    ax.plot(
        results[:, 0, 0],
```

(continues on next page)

(continued from previous page)

```
        results[:, 0, 1],
        "-o",
        markersize=7,
        color="r",
        alpha=0.5,
    )
    ax.set_xlim(-8, 8)
    ax.set_ylim(-4.3, 4.3)
```

6.4.1 Direct Search

Analogous to the golden section search for one-dimensional optimization, in direct search methods for multidimensional optimization the objective function values are only *compared* to each other. However, in contrast to the golden section search, they do not retain the convergence guarantee.

Perhaps the best known direct search method is the one of **Nelder and Mead**. To seek the minimum of a function $f : \mathbb{R}^n \to \mathbb{R}$, the function is first evaluated at $n+1$ starting points. These $n+1$ starting points form a *simplex* meaning that no three points are colinear (e.g. a simplex in two dimensions, has three points which form a triangle). A new point is generated along the straight line connecting the point with the highest function value (the *worst* point) and the centroid of the remaining n points. This new point then replaces the worst point and the process is repeated until convergence.

Direct search methods are especially useful for nonsmooth objective functions, for which few other methods are applicable, and they can be effective when n is small, but they tend to be quite expensive when n is larger than two or three. One advantage of direct search methods is that they can easily be parallelized.

6.4.2 Steepest Descent

As expected, greater use of the objective function and its derivatives leads to faster methods. The negative gradient of a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ points *downhill* and locally, $-\nabla f(\mathbf{x})$ is the direction of *steepest descent*. Thus, the negative gradient is a potentially fruitful direction in which to seek points having lower function values.

The maximum possible benefit from movement in any downhill direction is to attain the minimum of the objective function along that direction. For any fixed $\mathbf{x}$ and direction $\mathbf{s} = -\nabla f(\mathbf{x})$, we can define a function $\phi : \mathbb{R} \to \mathbb{R}$:

$$\phi(\alpha) = f(\mathbf{x} + \alpha\mathbf{s})$$

In this way the problem of minimizing the objective function f along the direction of $\mathbf{s}$ from $\mathbf{x}$ is seen to be a one-dimensional optimization problem that can be solved by one of the methods discussed in the previous section. Once a minimum is found in a certain direction, the negative gradient is computed at this new point and the process is repeated until convergence. This process of minimizing an objective function only along a fixed line in $\mathbb{R}^n$ is called a *line search*.

The steepest descent method is very reliable in that it can always make progress provided the gradient is nonzero. However, as the example below will illustrate the resulting iterations can zigzag back and forth, making very slow progress. In general, the convergence rate of steepest descent is only linear.

An implementation of the steepest descent method is shown below, together with an example in two dimensions.

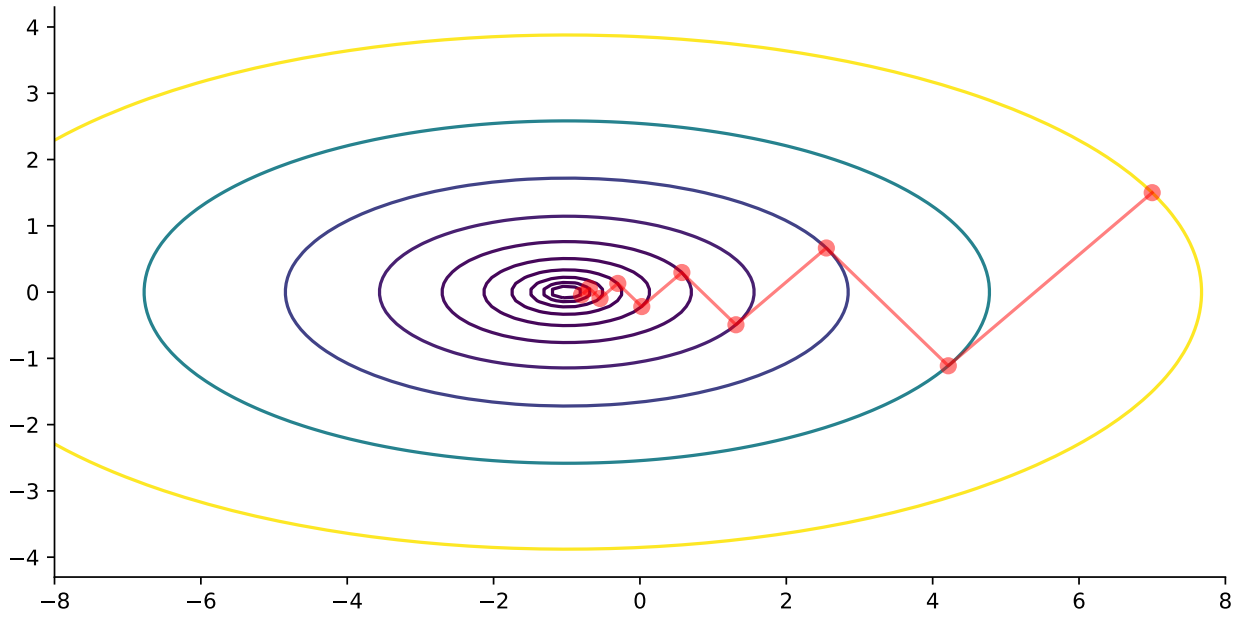
```
def my_steepest_descent(f, fp, x):
    """Illustrative implementation of the steepest descent method."""
    print("  i        f(x)")
    print("---  ----------")
    print(f"{0:3d}  {f(x):8.4e}")

    xk = []
    xk.append([x])

    for i in range(1, 10):
```

(continues on next page)

(continued from previous page)

```
        s = -fp(x)

        # line search
        res = optimize.minimize_scalar(
            (lambda alpha, x, s: f(x + alpha * s)), [0, 1], args=(x, s),␣
↪tol=1e-8
        )

        # step to minimum along line
        x = x + res.x * s
        xk.append([x])
        print(f"{i:3d}  {f(x):8.4e}")

    return np.array(xk)


results_sd = my_steepest_descent(func_multi, func_multi_p, x=np.array([7, 1.
↪5]))
plot_opt_trajectory(results_sd, "steepest")
```

```
  i        f(x)
---  ----------
  0  3.8125e+01
  1  1.7184e+01
  2  7.8978e+00
  3  3.7803e+00
  4  1.9545e+00
  5  1.1450e+00
  6  7.8599e-01
  7  6.2681e-01
  8  5.5623e-01
  9  5.2493e-01
```



6.4.3 Newton's Method

A broader view of the objective function can, once again, be gained from a local quadratic approximation, which can be obtained from a truncated Taylor series expansion

$$f(\mathbf{x}+\mathbf{s}) \approx f(\mathbf{x}) + \nabla f(\mathbf{x})^\intercal \mathbf{s} + \frac{1}{2}\mathbf{s}^\intercal \mathbf{H}_f(\mathbf{x})\mathbf{s},$$

where $\mathbf{H}_f(\mathbf{x})$ is the *Hessian matrix*. This quadratic function in $\mathbf{s}$ is minimized when

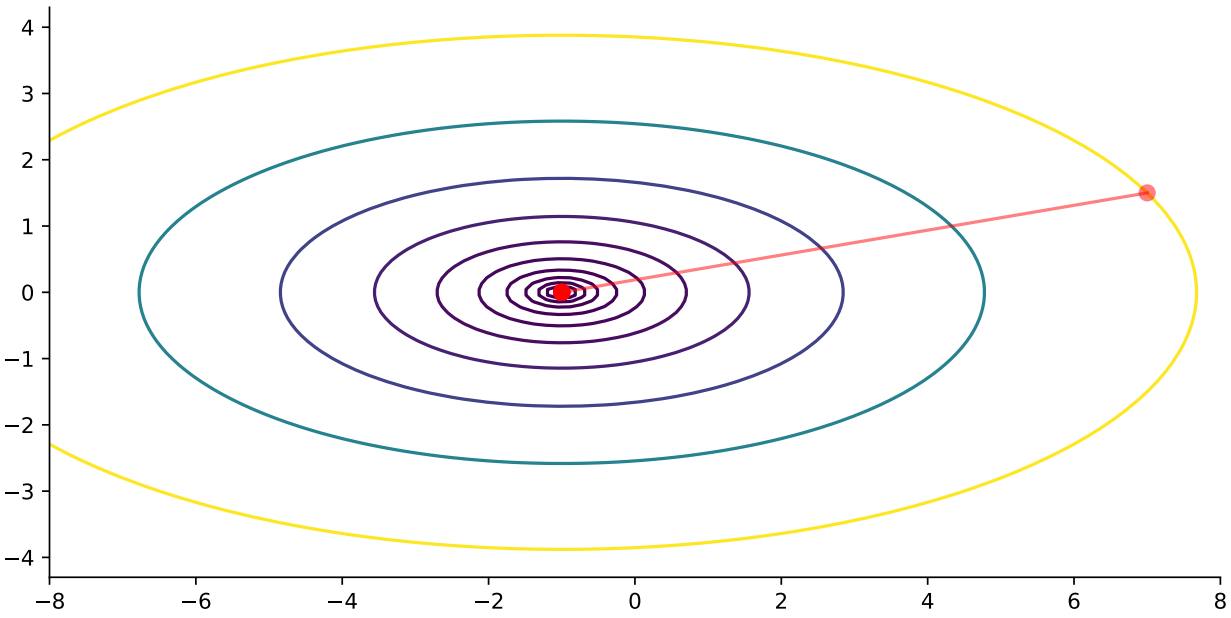
$$\mathbf{H}_f(\mathbf{x})\mathbf{s} = -\nabla f(\mathbf{x}).$$

The convergence rate of Newton's method for unconstrained optimization is normally quadratic but the method is unreliable unless started close enough to the solution. While Newton's method does not require a line search, it may still be advisable to perform a line search along the direction of the Newton step in order to make the method more robust.

Below, Newton's method is applied to the same two-dimensional example as before.

```
def my_newton_method_nd(f, fp, fpp, x):
    """Illustrative implementation of the multi-dimensional Newton method."""
    print("  i        f(x)")
    print("---  ----------")
    print(f"{0:3d}  {f(x):8.4e}")

    xk = []
    xk.append([x])

    for i in range(1, 10):
        # solve linear system
        hessian = fpp(x)
        s = linalg.solve(hessian, -fp(x))

        # make step
        x = x + s
        xk.append([x])
        print(f"{i:3d}  {f(x):8.4e}")

    return np.array(xk)


results_nm = my_newton_method_nd(
    func_multi, func_multi_p, func_multi_pp, x=np.array([7, 1.5])
)
plot_opt_trajectory(results_nm, "newton_nd")
```

```
  i        f(x)
---  ----------
  0  3.8125e+01
  1  5.0000e-01
  2  5.0000e-01
  3  5.0000e-01
  4  5.0000e-01
  5  5.0000e-01
  6  5.0000e-01
  7  5.0000e-01
  8  5.0000e-01
  9  5.0000e-01
```



6.4.4 Quasi-Newton Methods

Newton's method usually converges very rapidly once it nears a solution, but it requires a substantial amount of work per iteration. Specifically, for a problem with a dense Hessian matrix, each iteration requires $\mathcal{O}(n^2)$ scalar function evaluations to form the gradient and the Hessian matrix while $\mathcal{O}(n^3)$ arithmetic operations are required to solve the linear system for the Newton step. Many variants of Newton's method have been developed to reduce its overhead or improve its reliability, or both. These *quasi-Newton methods* have the general form

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{B}_k^{-1} \nabla f(\mathbf{x}_k)$$

where α_k is a line search parameter and $\mathbf{B}_k$ is some approximation of the Hessian matrix obtained in any number of ways, including secant updating, finite differences, periodic reevaluation, or neglecting some terms in the true Hessian of the objective function.

Many quasi-Newton methods are more robust than the pure Newton method and have considerably lower overhead per iteration yet remain superlinear (though not quadratic).

6.4.5 Secant Updating Methods

Several secant updating formulas for unconstrained minimization have been developed that not only preserve symmetry in the approximate Hessian matrix but also preserve positive definiteness. Symmetry reduces the amount of work and storage required by about half, and positive definiteness guarantees that the resulting quasi-Newton step will be a descent direction. In practice, a factorization of $\mathbf{B}_k$ is updated rather than $\mathbf{B}_k$ itself, so that the linear system for the quasi-Newton step can be solved at a cost per iteration of $\mathcal{O}(n^2)$ rather than $\mathcal{O}(n^3)$ operations.

Unlike Newton's method for optimization, no second derivatives are required. And most of these methods are often started with $\mathbf{B}_0 = \mathbf{I}$, which means the first step is along the negative gradient (i.e. along the direction of steepest descent) and then second derivative information is gradually built up in the approximate Hessian matrix by updating over successive iterations.

One of the most effective secant updating methods for optimization is called BFGS of which an implementation is shown below.

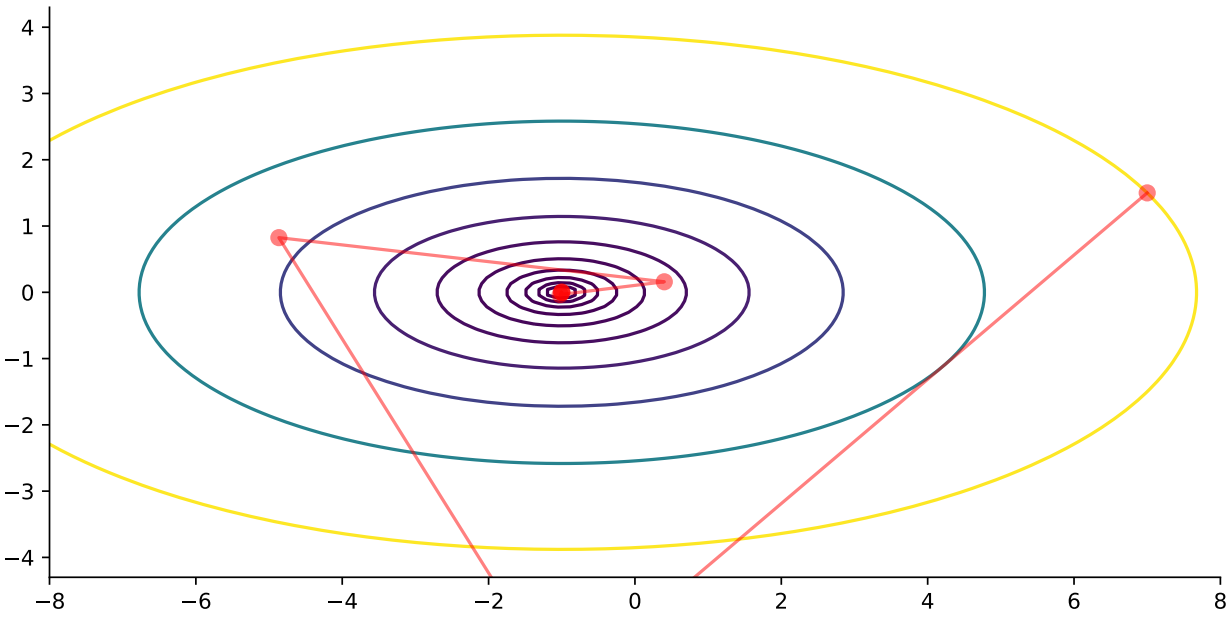
```
def my_bfgs(f, fp, B, x):
    """Illustrative implementation of the BFGS Quasi Newton method."""
```

(continues on next page)

(continued from previous page)

```
    print("  i        f(x)")
    print("---  ----------")
    print(f"{0:3d}  {f(x):8.4e}")

    xk = []
    xk.append([x])
    tol = 1e-8
    y = 1
    s = 1
    i = 0

    while (np.dot(y, s)) >= tol:
        i += 1
        # solve linear system + make step (cfr. Newton's Method)
        s = linalg.solve(B, -fp(x))
        x_new = x + s

        # update approximation of Hessian
        y = fp(x_new) - fp(x)
        B = (
            B
            + (np.outer(y, y) / np.dot(y, s))
            - (np.dot(np.outer(np.dot(B, s), s), B) / multi_dot([s, B, s]))
        )  # update B with BFGS formula

        x = x_new
        xk.append([x])
        print(f"{i:3d}  {f(x):8.4e}")

    return np.array(xk)


# initial guess B = I
results_bfgs = my_bfgs(
    func_multi, func_multi_p, B=np.array([[1, 0], [0, 1]]), x=np.array([7, 1.
 ↪5])
)
plot_opt_trajectory(results_bfgs, "bfgs")
```

```
  i        f(x)
---  ----------
  0  3.8125e+01
  1  9.0500e+01
  2  9.6728e+00
  3  1.5446e+00
  4  5.0314e-01
  5  5.0001e-01
  6  5.0000e-01
  7  5.0000e-01
```



Note that the increase in function value on the first iteration in the example above could have been avoided by using a line search. Such a line search can also be used to enhance the effectiveness of the method. For a quadratic expression it can be stated that if an exact line search is added to every iteration of the BFGS method, it will terminate at the exact solution after at most n iterations, where n is the dimension of the problem.

6.4.6 Conjugate Gradient Method

The conjugate gradient method is another alternative to Newton's method that does not require explicit second derivatives. Indeed, unlike secant updating methods, the conjugate gradient method does not even store an approximation to the Hessian matrix, which makes it especially **suitable for very large problems**.

As the name suggests, the conjugate gradient method also uses gradients, but in contrast to the steepest descent method it avoids repeatedly searching in the same directions by modifying the new gradient at each step to remove components in previous directions. The resulting sequence of *conjugate* (i.e. orthogonal in the inner product $(\mathbf{x}, \mathbf{y}) = \mathbf{x}^\intercal \mathbf{H}_f \mathbf{y}$ search directions implicitly accumulates information about the Hessian matrix as iterations proceed.

Theoretically, the conjugate gradient method is exact after at most n iterations for a quadratic objective function in n dimensions, but it is usually quite effective for more general unconstrained optimization problems as well. It is common to restart the algorithm after every n iterations by restarting to use the negative gradient at the current point.

In the following example, an implementation of the conjugate gradient method is applied to minimize a function in two dimensions.

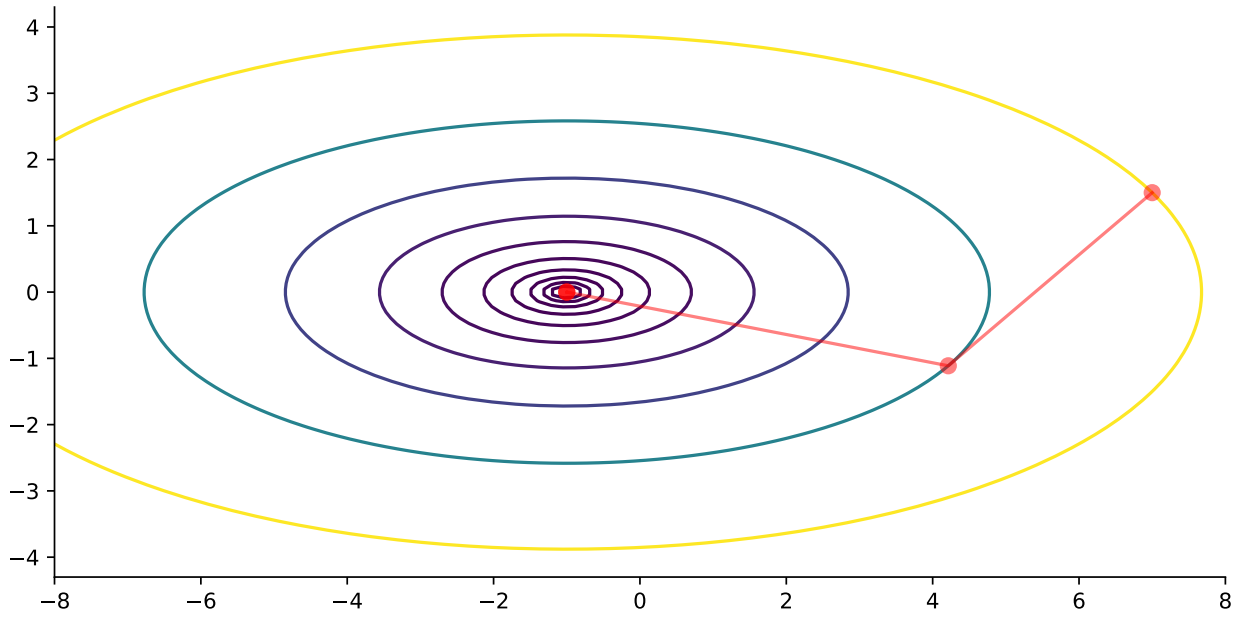
```
def my_conjugate_gradient(f, fp, x):
    """Illustrative implementation of the conjugate gradient method."""
    print("  i        f(x)")
    print("---  ----------")
    print(f"{0:3d}  {f(x):8.4e}")

    xk = []
    xk.append([x])

    # initialize parameters
    g = fp(x)
    s = -g
    for i in range(1, 5):
```

(continues on next page)

(continued from previous page)

```
        # line search + make step
        res = optimize.minimize_scalar(
            (lambda alpha, x, s: f(x + alpha * s)), [0, 1], args=(x, s),␣
↪tol=1e-8
        )
        x_new = x + res.x * s

        g_new = fp(x_new)
        # Polak-Ribiere with built-in reset
        b = max(np.inner(g_new - g, g_new) / np.inner(g, g), 0)
        s = -g_new + b * s

        x = x_new
        g = g_new

        xk.append([x])
        print(f"{i:3d}  {f(x):8.4e}")

    return np.array(xk)


results_cg = my_conjugate_gradient(func_multi, func_multi_p, x=np.array([7, 1.
↪5]))
plot_opt_trajectory(results_cg, "cg")
```

```
  i        f(x)
---  ----------
  0  3.8125e+01
  1  1.7184e+01
  2  5.0000e-01
  3  5.0000e-01
  4  5.0000e-01
```



6.5 Non-linear Least Squares

Least squares data fitting can be viewed as an optimization problem. Given data points $(t_i, y_i), i = 1, \ldots, m$, we wish to find the vector $\mathbf{x} \in \mathbb{R}^n$ of parameters that gives the best fit to the model function $f(t, \mathbf{x})$, where $f : \mathbb{R}^{n+1} \to \mathbb{R}$. Previously, we only considered cases in which the model function f was linear in the components of $\mathbf{x}$ but now we are in a position to consider *nonlinear least squares* as a special case of nonlinear optimization.

If we define the components of the *residual* function $\mathbf{r} : \mathbb{R}^n \to \mathbb{R}^m$ by

$$r_i(\mathbf{x}) = y_i - f(t_i, \mathbf{x}), \quad i = 1, \ldots, m,$$

then we wish to minimize the function

$$\phi(\mathbf{x}) = \frac{1}{2}\mathbf{r}(\mathbf{x})^\intercal \mathbf{r}(\mathbf{x})$$

i.e. the sum of squares of the residual components (the factor 1/2 is inserted for later convenience and has no effect on the optimal value for $\mathbf{x}$). If we apply Newton's method and $\mathbf{x}_k$ is an approximate solution, then the Newton step $\mathbf{s}_k$ is given by the linear system

$$\mathbf{H}_\phi(\mathbf{x}_k)\mathbf{s}_k = -\nabla\phi(\mathbf{x}_k)$$

where the gradient vector and Hessian matrix of ϕ are given by

$$\nabla\phi(\mathbf{x}) = \mathbf{J}^\intercal(\mathbf{x})\mathbf{r}(\mathbf{x})$$

and

$$\mathbf{H}_\phi(\mathbf{x}) = \mathbf{J}^\intercal(\mathbf{x})\mathbf{J}(\mathbf{x}) + \sum_{i=1}^{m} r_i(\mathbf{x})\mathbf{H}_{r_i}(\mathbf{x})$$

in which $\mathbf{J}^\intercal(\mathbf{x})$ is the Jacobian matrix of $\mathbf{r}(\mathbf{x})$, and $\mathbf{H}_{r_i}(\mathbf{x})$ denotes the Hessian matrix of the component function $r_i(\mathbf{x})$.

The Newton step $\mathbf{s}_k$ is thus given by the linear system

$$\left(\mathbf{J}^\intercal(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k) + \sum_{i=1}^{m} r_i(\mathbf{x}_k)\mathbf{H}_{r_i}(\mathbf{x}_k)\right)\mathbf{s}_k = -\mathbf{J}^\intercal(\mathbf{x}_k)\mathbf{r}(\mathbf{x}_k)$$

The m Hessian matrices $\mathbf{H}_{r_i}$ of the residual components are usually inconvenient and expensive to compute. Fortunately, we can exploit the special structure of this problem to avoid computing them in most cases, as we will see next.

6.5.1 Gauss-Newton Method

Note that in $\mathbf{H}_\phi$ each of the Hessian matrices $\mathbf{H}_{r_i}$ is multiplied by the corresponding residual component r_i, which should be small at a solution, provided that the model function fits the data reasonably well. This observation motivates the *Gauss-Newton method* for nonlinear least squares in which the terms involving $\mathbf{H}_{r_i}$ are dropped from the Hessian and the linear system

$$\left(\mathbf{J}^\intercal(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k)\right)\mathbf{s}_k = -\mathbf{J}^\intercal(\mathbf{x}_k)\mathbf{r}(\mathbf{x}_k)$$

determines an approximate Newton step $\mathbf{s}_k$ at each iteration.

We recognize this system as the normal equations for the $m \times n$ linear least squares problem

$$\mathbf{J}(\mathbf{x}_k)\mathbf{s}_k \cong -\mathbf{r}(\mathbf{x}_k)$$

which can be solved more reliably by orthogonal factorization of $\mathbf{J}(\mathbf{x}_k)$. The next approximate solution is then given by $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$ and the process is repeated until convergence. In effect, the Gauss-Newton method replaces a nonlinear least squares problem by a sequence of linear least squares problems whose solutions converge to the solution of the original nonlinear problem.

If the residual components at the solution are relatively large, then the terms omitted from the Hessian matrix may not be negligible, in which case the Gauss-Newton approximation may be inaccurate and convergence is no longer guaranteed. In such cases, it may be best to use a general nonlinear optimization method that takes into account the full Hessian matrix.

Example: radioactive decay

As an example the Gauss-Newton method will be used to fit the time dependency of the measured activity of a radioactive Fermium-246 sample. The experiment yielded the following four data points

t (s)	A (s^{-1})
0.0	514
1.0	303
2.0	201
3.0	113

and the model prescription has the form

$$f(t, \mathbf{x}) = x_1 e^{x_2 t} = A_0 e^{-\lambda t},$$

where $x_1 = A_0$ is the initial activity of the sample and $x_2 = -\lambda$ is the decay constant of the radioactive isotope which is closely related to the half-life constant.

For this model function, the entries of the Jacobian matrix of the residual function $\mathbf{r}$ are given by

$$\{\mathbf{J}(\mathbf{x})\}_{i,1} = \frac{\partial r_i(\mathbf{x})}{\partial x_1} = -e^{x_2 t_i}, \qquad \{\mathbf{J}(\mathbf{x})\}_{i,2} = \frac{\partial r_i(\mathbf{x})}{\partial x_2} = -x_1 t_i e^{x_2 t_i}$$

where the minus sign originates from the definition of $\mathbf{r}$ ($= A_i - f(t_i, \mathbf{x})$) and $i = 1, \ldots, 4$. If we start with $\mathbf{x}_0 = [480, \ -0.4]^\intercal$ as the initial guess, then the linear least squares problem to be solved for the Gauss-Newton step $\mathbf{s}_0$ is

$$\mathbf{J}(\mathbf{x}_0)\mathbf{s}_0 = \begin{bmatrix} -1.00 & 0 \\ -0.67 & -322 \\ -0.45 & -431 \\ -0.30 & -434 \end{bmatrix} \mathbf{s}_0 \cong \begin{bmatrix} -34.00 \\ 18.75 \\ 14.68 \\ 31.57 \end{bmatrix} = -\mathbf{r}(\mathbf{x}_0)$$

The least squares solution of this system is $\mathbf{s}_0 = [30.75, \ -0.089]^\intercal$ and hence we take $\mathbf{x}_1 = \mathbf{x}_0 + \mathbf{s}_0 = [510.75, \ -0.489]^\intercal$ for the next step. As illustrated in the implementation below, the system converges after several iterations. From the decay constant $\lambda = -x_2$ the half-life of Fermium-246 can be calculated according to

$$T_{1/2} = \frac{\ln(2)}{\lambda} = \frac{\ln(2)}{0.494} = 1.4s$$

which is indeed very close to the actual half-life of 1.5 s.

```
def radioactiveModel(t, x):
    """model function"""
    return x[0] * np.exp(x[1] * t)


def jacobian(t, x):
    """Jacobian matrix of r(x)"""
    J1 = -np.exp(x[1] * t)
    J2 = -x[0] * t * np.exp(x[1] * t)

    return np.array([J1, J2]).T
```

(continues on next page)

(continued from previous page)

```
def residual(t, y, x):
    return np.array(y - radioactiveModel(t, x))
```

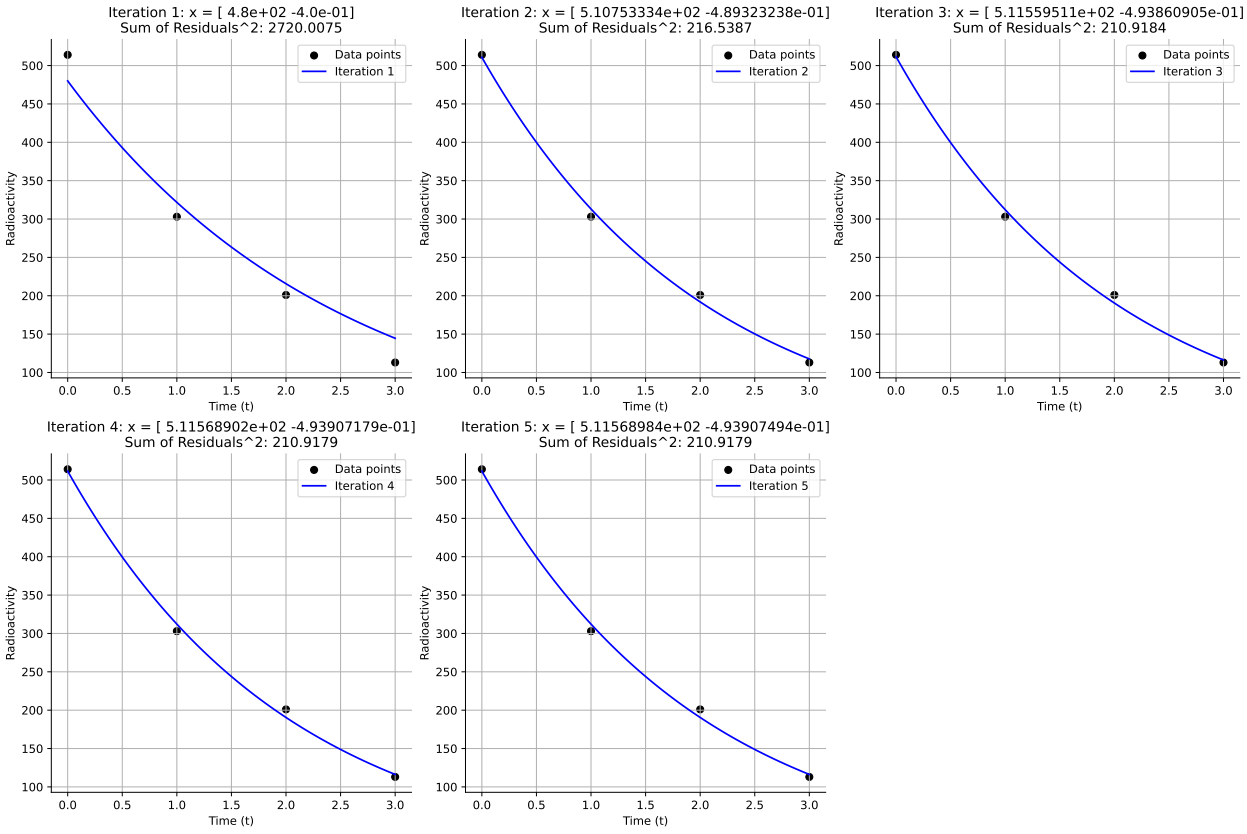
```
# Nonlinear least squares fitting using Gauss-Newton method
def gauss_newton_static_plot(initial_guess):
    t = np.arange(0, 4)
    y = np.array([514, 303, 201, 113])
    x = np.array(initial_guess)

    iterations = []
    models = []
    residuals = []

    print("Iteration    Parameters (x1, x2)              Squared Sum of␣
 ↪Residuals")
    print("-------------------------------------------------------------------
 ↪----")

    for i in range(5):  # Perform 5 iterations
        iterations.append(x.copy())
        models.append(radioactiveModel(t, x))
        r = residual(t, y, x)
        residuals.append(np.sum(r**2))  # Calculate squared sum of residuals

        print(
            f"{i + 1:>4}        {x[0]:>10.4f}, {x[1]:>10.4f}         "
            f"{residuals[-1]:>20.4f}"
        )

        J = jacobian(t, x)
        p, _, _, _ = lstsq(J, -r, rcond=None)
        x = x + p

    # Generate the static plot
    plt.close("gauss_newton")
    fig, axs = plt.subplots(2, 3, figsize=(15, 10), num="gauss_newton")
    axs = axs.flatten()

    t_fine = np.linspace(0, 3, 100)
    for i, ax in enumerate(axs[:5]):
        ax.scatter(t, y, label="Data points", color="black")
        ax.plot(
            t_fine,
            radioactiveModel(t_fine, iterations[i]),
            label=f"Iteration {i + 1}",
            color="blue",
        )
        ax.set_title(
            f"Iteration {i + 1}: x = {iterations[i]}\n"
            f"Sum of Residuals^2: {residuals[i]:.4f}"
        )
        ax.set_xlabel("Time (t)")
        ax.set_ylabel("Radioactivity")
        ax.legend()
```

(continues on next page)

(continued from previous page)

```
        ax.grid()

    # Hide the last unused subplot
    axs[-1].axis("off")


# Call the function to generate the static plot
gauss_newton_static_plot([480, -0.4])
```

```
Iteration    Parameters (x1, x2)               Squared Sum of Residuals
-------------------------------------------------------------------------
   1          480.0000,    -0.4000                    2720.0075
   2          510.7533,    -0.4893                     216.5387
   3          511.5595,    -0.4939                     210.9184
   4          511.5689,    -0.4939                     210.9179
   5          511.5690,    -0.4939                     210.9179
```



Like all methods based on Newton's method, the Gauss-Newton method for solving nonlinear least squares problems may fail to converge unless it is started sufficiently close to the solution. A line search can be used to improve its robustness, but additional modifications may be necessary to ensure that the computed step $\mathbf{s}_k$ is a descent direction when far from the solution.

6.5.2 Levenberg-Marquardt Method

The *Levenberg-Marquardt method* is a useful alternative when the Gauss-Newton method yields an ill-conditioned or rank-deficient linear least squares subproblem. At each iteration of this method, the linear system for the step $\mathbf{s}_k$ is of

the form

$$\left(\mathbf{J}^{\mathsf{T}}(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k) + \mu_k \mathbf{I}\right)\mathbf{s}_k = -\mathbf{J}^{\mathsf{T}}(\mathbf{x}_k)\mathbf{r}(\mathbf{x}_k)$$

where μ_k is a nonnegative scalar parameter chosen by some strategy. The corresponding linear least squares problem to be solved is

$$\begin{bmatrix} \mathbf{J}(\mathbf{x}_k) \\ \sqrt{\mu_k}\,\mathbf{I} \end{bmatrix} \mathbf{s}_k \cong \begin{bmatrix} -\mathbf{r}(\mathbf{x}_k) \\ \mathbf{0} \end{bmatrix}$$

This method can be interpreted as replacing the terms omitted from the true Hessian by a scalar multiple of the identity matrix or as using a weighted combination of the Gauss-Newton step and the steepest descent direction. With a suitable strategy for choosing the parameter μ_k, typically based on a trust-region approach, the Levenberg-Marquardt method can be very robust in practice, and it forms the basis for several effective software packages for solving nonlinear least squares problems.

6.6 Constrained optimization

So far, we only considered minima that occur at an interior point of the feasible set S. For **constrained optimization problems**, the minimum is often located outside the feasible set, meaning that the solution of the constrained optimization problem occurs on the boundary of the feasible set. The principles to find a minimum remains the same: a minimum occurs at $\mathbf{x}^* \in S$ when there is no downhill direction starting from $\mathbf{x}^*$, considering only **feasible directions**, i.e. directions for which the constraints continue to be satisfied.

In general, the constraints can be linear or nonlinear and can be subdivided into the following two categories:

- **Equality constraints** which are of the general form $\mathbf{g}(\mathbf{x}) = \mathbf{0}$
- **Inequality constraints** which are of the general form $\mathbf{h}(\mathbf{x}) \leq \mathbf{0}$

Inequality constraints may be irrelevant to the solution and a given inequality constraint $h_i(\mathbf{x}) \leq 0$ is said to be *active* or *binding* at a feasible point $\mathbf{x} \in \mathbf{S}$ if $h_i(\mathbf{x}) = 0$. Naturally, equality constraints are always active.

Both equality-constrained and inequality-constrained problems can be solved using *Lagrange multipliers* although the optimality conditions become more complicated when inequality constraints are involved.

Example: the Rosenbrock function

As an example, let us consider minimization of the Rosenbrock function

$$\min_{x_0,x_1} \left(100\,(x_1 - x_0^2)^2 + (1 - x_0)^2\right)$$

under the condition that the following constraints apply:

$$\begin{aligned} x_0 + 2x_1 &\leq 1 \\ x_0^2 + x_1 &\leq 1 \\ x_0^2 - x_1 &\leq 1 \\ 2x_0 + x_1 &= 1 \\ 0 \leq x_0 &\leq 1 \\ -0.5 \leq x_1 &\leq 2 \end{aligned}$$

This constrained optimization problem has a unique solution $\mathbf{x}^* = [0.4149 \quad 0.1701]^{\mathsf{T}}$ for which only the fourth constraint is active. Note, however, that the Rosenbrock function has a global minimum at $\mathbf{x}' = [1\ 1]^{\mathsf{T}}$ but that this point is not a feasible point $\left(\mathbf{x}' \notin \mathbf{S}\right)$ because it violates multiple constraints.

6.6.1 The trust-region constrained algorithm

The minimize function of `scipy.optimize` provides several algorithms for constrained minimization. One of which is the trust-region constrained algorithm, which deals with constrained minimization problems of the form:

$$\begin{aligned} &\min_{\mathbf{x}} f(\mathbf{x}) \\ &\text{subject to:} \quad \mathbf{c}_l \leq \mathbf{c}(\mathbf{x}) \leq \mathbf{c}_u \\ &\qquad\qquad\quad \mathbf{x}_l \leq \mathbf{x} \leq \mathbf{x}_u \end{aligned}$$

When $c_{l,j} = c_{u,j}$ the method reads the jth constraint as an equality constraint and deals with it accordingly. Besides that, one-sided constraints can be specified by setting the upper (u) or lower (l) bound to `np.inf` with the appropriate sign. As an illustration, this method will be applied to the Rosenbrock example.

Example: the trust-region constrained algorithm The method requires the constraints to be defined as a sequence of Bounds, LinearConstraint and NonlinearConstraint objects. Therefore the first step is to classify the constraints from the above example accordingly.

- There are two *bound constraints*, namely $0 \leq x_0 \leq 1$ and $-0.5 \leq x_1 \leq 2$.
- There are two *linear constraints*, namely $x_0 + 2x_1 \leq 1$ and $2x_0 + x_1 = 1$
- There are two *nonlinear constraints*, namely $x_0^2 + x_1 \leq 1$ and $x_0^2 - x_1 \leq 1$.

The linear constraints can be written in the standard format

$$\begin{bmatrix} -\infty \\ 1 \end{bmatrix} \leq \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \leq \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Whereas the nonlinear constraints can be written as

$$\begin{bmatrix} -\infty \\ -\infty \end{bmatrix} \leq \begin{bmatrix} x_0^2 + x_1 \\ x_0^2 - x_1 \end{bmatrix} = \mathbf{c}(\mathbf{x}) \leq \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

The NonlinearConstraints object also takes the Jacobian matrix and a linear combination of Hessians of $\mathbf{c}(\mathbf{x})$ as optional arguments. If no Jacobian matrix or Hessian is passed, the gradient will be estimated using 2-point finite difference estimation with an absolute step size while the Hessian will be estimated using one of the quasi-Newton strategies. However, for this example they can be calculated with relative ease which gives:

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} 2x_0 & 1 \\ 2x_0 & -1 \end{bmatrix}$$

$$\mathbf{H}(\mathbf{x}, \mathbf{v}) = \sum_i v_i \nabla^2 c_i(\mathbf{x}) = v_0 \begin{bmatrix} 2 & 0 \\ 0 & 0 \end{bmatrix} + v_1 \begin{bmatrix} 2 & 0 \\ 0 & 0 \end{bmatrix}$$

With this information it is now possible to solve the constrained optimization problem with `scipy.optimize.minimize` as shown below.

```
def rosen(x):
    return sum(100.0 * (x[1:] - x[:-1] ** 2.0) ** 2.0 + (1 - x[:-1]) ** 2.0)


def cons_c(x):
    return [x[0] ** 2 + x[1], x[0] ** 2 - x[1]]


def cons_J(x):
    return [[2 * x[0], 1], [2 * x[0], -1]]
```

(continues on next page)

(continued from previous page)

```
def cons_H(x, v):
    return v[0] * np.array([[2, 0], [0, 0]]) + v[1] * np.array([[2, 0], [0,␣
 ↪0]])


# Initialize constraint objects
bounds = optimize.Bounds([0, -0.5], [1.0, 2.0])
linear_constraint = optimize.LinearConstraint([[1, 2], [2, 1]], [-np.inf, 1],␣
 ↪[1, 1])
nonlinear_constraint = optimize.NonlinearConstraint(
    cons_c, -np.inf, 1, jac=cons_J, hess=cons_H
)

# Solve the optimization
x0 = np.array([0.5, 0])  # initial guess
res = optimize.minimize(
    rosen,
    x0,
    method="trust-constr",
    constraints=[linear_constraint, nonlinear_constraint],
    bounds=bounds,
)

print("Result: ", res.x)
```

```
Result:  [0.41494531 0.17010937]
```

which indeed corresponds to the expected solution of $\mathbf{x}^* = [0.4149 \quad 0.1701]^\intercal$

```
def plot_rosen(less_busy=False):
    """Plot Rosenbrock function with constraints and (constrained) minimum."""

    # Define the Rosenbrock function and its constraints
    def Rosen(x, y):
        return (1 - x) ** 2 + 100.0 * (y - x**2) ** 2

    def constraint_1(x, y):
        return x - 2 * y - 1  # Reformulated as a <= 0

    def constraint_2(x, y):
        return x**2 + y - 1  # Reformulated as a <= 0

    def constraint_3(x, y):
        return x**2 - y - 1  # Reformulated as a <= 0

    def constraint_4(x, y):
        return 2 * x + y - 1  # Reformulated as a <= 0

    def constraint_5a(x, y):
        return x - 1  # Reformulated as a <= 0

    def constraint_5b(x, y):
        return -x  # Reformulated as a <= 0
```

(continues on next page)

(continued from previous page)

```
    def constraint_6a(x, y):
        return y - 1  # Reformulated as a <= 0

    def constraint_6b(x, y):
        return -y - 0.5  # Reformulated as a <= 0

    # Define grid
    x = np.linspace(-2.0, 2.0, 300)
    y = np.linspace(-1.0, 3.0, 300)
    xx, yy = np.meshgrid(x, y)

    # Plot Rosenbrock function
    plt.close("rosen")
    fig, ax = plt.subplots(num="rosen")
    cm = ax.contourf(
        xx, yy, Rosen(xx, yy), 20, cmap="viridis", alpha=0.7 if less_busy␣
↪else 1
    )
    plt.colorbar(cm)

    # Add contour lines
    ax.contour(xx, yy, np.log(Rosen(xx, yy)), 20, colors="black",␣
↪linewidths=1.0)

    # Plot constraints
    constraints = [
        constraint_1,
        constraint_2,
        constraint_3,
        constraint_4,
        constraint_5a,
        constraint_5b,
        constraint_6a,
        constraint_6b,
    ]

    if less_busy:
        # Less busy: shade regions outside feasible set
        for constraint in constraints:
            ax.contourf(
                xx,
                yy,
                constraint(xx, yy),
                levels=[0, np.inf],
                colors=["gray"],
                alpha=1,
            )
    else:
        # Full plot: add detailed constraint lines
        for constraint in constraints:
            ax.contour(
                xx,
                yy,
```

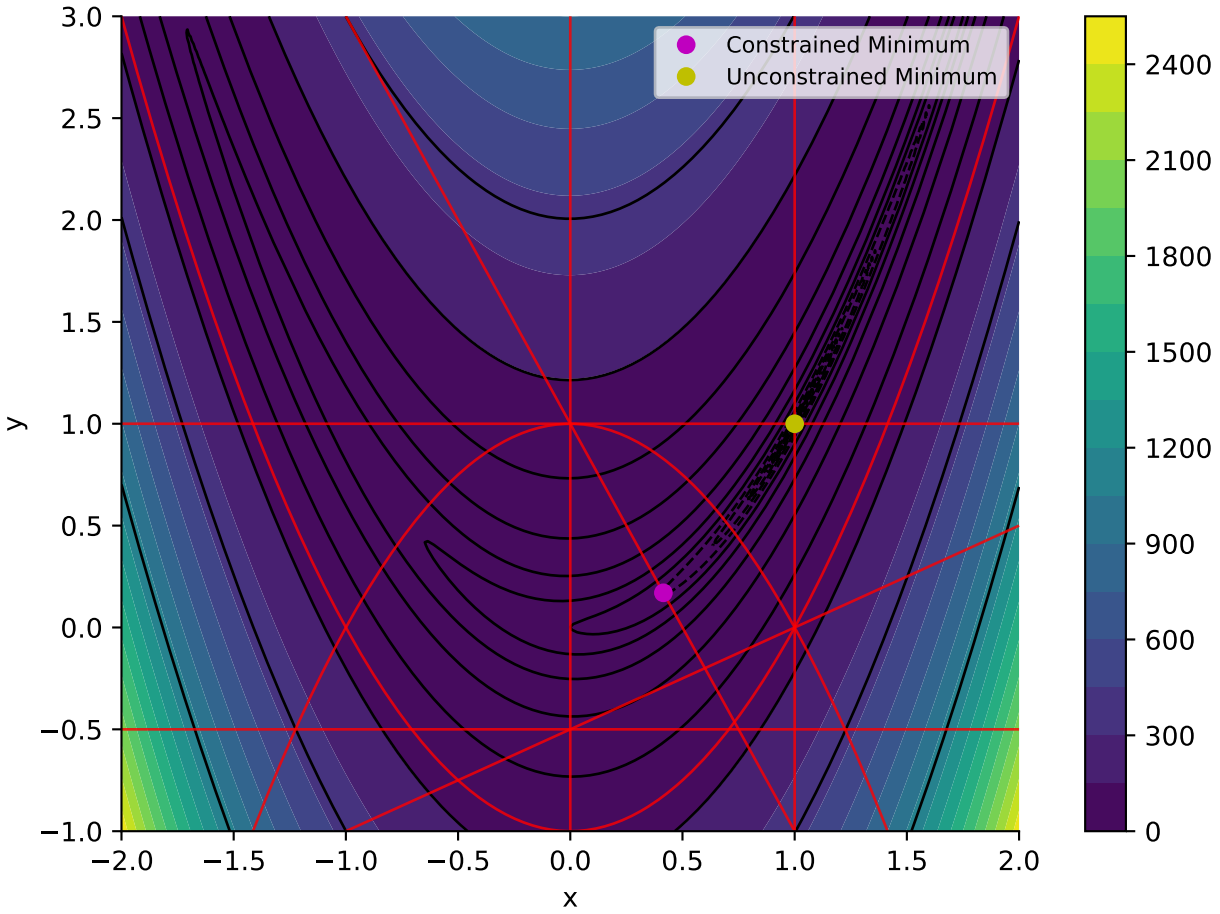
(continues on next page)

(continued from previous page)

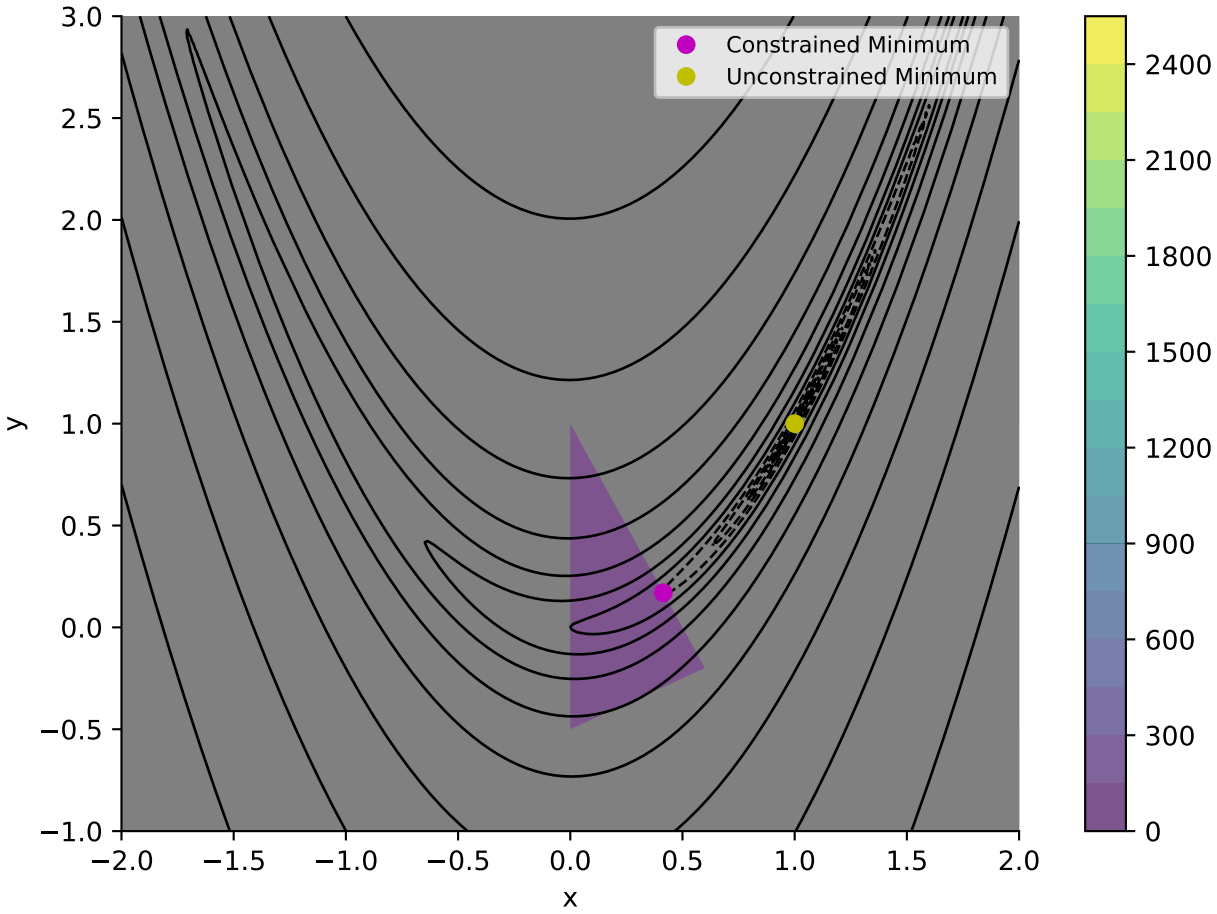
```
                constraint(xx, yy),
                levels=[0],
                colors="red",
                alpha=0.8,
                linewidths=1.0,
            )

    # Mark points
    x_min, y_min = res.x
    ax.plot(x_min, y_min, "mo", label="Constrained Minimum")
    ax.plot(1, 1, "yo", label="Unconstrained Minimum")

    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.legend(loc=0, fontsize="small")


# Objective function and constraints
def rosen(x):
    return sum(100.0 * (x[1:] - x[:-1] ** 2.0) ** 2.0 + (1 - x[:-1]) ** 2.0)


def cons_c(x):
    return [x[0] ** 2 + x[1], x[0] ** 2 - x[1]]


def cons_J(x):
    return [[2 * x[0], 1], [2 * x[0], -1]]


def cons_H(x, v):
    return v[0] * np.array([[2, 0], [0, 0]]) + v[1] * np.array([[2, 0], [0,␣
 ↪0]])


# Initialize constraint objects
bounds = optimize.Bounds([0, -0.5], [1.0, 2.0])
linear_constraint = optimize.LinearConstraint([[1, 2], [2, 1]], [-np.inf, 1],␣
 ↪[1, 1])
nonlinear_constraint = optimize.NonlinearConstraint(
    cons_c, -np.inf, 1, jac=cons_J, hess=cons_H
)

# Solve the optimization
x0 = np.array([0.5, 0])  # initial guess
res = optimize.minimize(
    rosen,
    x0,
    method="trust-constr",
    constraints=[linear_constraint, nonlinear_constraint],
    bounds=bounds,
)

print("Result: ", res.x)
```

```
Result:  [0.41494531 0.17010937]
```

```
plot_rosen(less_busy=False);  # Full plot
```




```
plot_rosen(less_busy=True)  # Less busy plot
```


Constrained Minimum
Unconstrained Minimum
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
y
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x
2400
2100
1800
1500
1200
900
600
300
0

CHAPTER 7

Interpolation

```
import ipywidgets as widgets
import matplotlib.pyplot as plt
import numpy as np
from scipy import interpolate, linalg
```

7.1 Introduction

Starting from a set of measurement (or simulated) data, you'll often want to perform further analysis, which requires you to use intermediate (or interpolated) data points for which you don't know the exact value.

Depending on the situation, it might be appropriate to use a least-squares approximation to fit your data to an underlying physical model. In other situations, it is better to try to fit your data exactly, using interpolation.

The main difference between an approximate fitting procedure and interpolation is that the function must match the given data values exactly. **Interpolation** thus means fitting some function to given data so that the function has the same values as the given data.

There are many different purposes for which you might want to use interpolation:

- Plotting a smooth curve through discrete data points
- Reading between the lines of a table
- Differentiating or integrating tabular data
- Evaluating a mathematical function quickly and easily
- Replacing a complicated function by a simple one

For given data $(t_i, y_i), i = 1, \ldots, m,$

with $t_1 < t_2 < \ldots < t_m$, we seek a function $f : \mathbb{R} \to \mathbb{R}$ such that

$f(t_i) = y_i, i = 1, \ldots, m,$

We call f an **interpolating function** or **interpolant**.

The question of existence and uniqueness of an interpolant comes down to matching the number of parameters in the interpolant to the number of data points to be fit: if there are too few parameters, then the interpolant does not exist; if there are too many parameters, then the interpolant is not unique.

For a given set of data points $(t_i, y_i), i = 1, \ldots, m,$, an interpolant is chosen from the space of functions spanned by a suitable set of **basis functions** $\phi_1(t), \ldots, \phi_n(t)$.

The interpolating function f is therefore expressed as a linear combination of these basis functions

$$f(t) = \sum_{j=1}^{n} x_j \phi_j(t)$$

where the parameters x_j are to be determined. Requiring that f interpolate the data (t_i, y_i) means that

$$f(t_i) = \sum_{j=1}^{n} x_j \phi_j(t_i) = y_i$$

which is a system of linear equations that we can write in matrix form as $\mathbf{Ax} = \mathbf{y}$, where the entries of the **basis matrix** $\mathbf{A}$ are given by $a_{ij} = \phi_j(t_i)$, the components of the right-hand-side vector $\mathbf{y}$ are the known data points y_i, and the components of the vector $\mathbf{x}$ the unknown parameters x_j we want to determine.

7.2 Polynomial interpolation of discrete data

We denote by $\mathbb{P}_k (k \geq 0)$ the set of polynomials with degree lower or equal than k. $\mathbb{P}_k$ then is a vector space of dimension $k + 1$. The particular choice of basis functions has a tremendous effect on the computational cost.

7.2.1 Monomial basis

To interpolate n data points, we choose $k = n - 1$ so that the dimension of the space will match the number of data points. An obvious basis for $\mathbb{P}_{n-1}$ is given by the first n **monomials**

$$\phi_j(t) = t^{j-1}$$

for which a given polynomial $p_{n-1} \in \mathbb{P}_{n-1}$ has the form

$$p_{n-1}(t) = x_1 + x_2 t + \cdots + x_n t^{n-1}$$

The system of equations we want to solve is

$$\mathbf{Ax} = \begin{bmatrix} 1 & t_1 & t_1^2 & \cdots & t_1^{n-1} \\ 1 & t_2 & t_2^2 & \cdots & t_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & t_n & t_n^2 & \cdots & t_n^{n-1} \end{bmatrix} \begin{bmatrix} x1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \mathbf{y}$$

> **Example**
>
> We will determine the polynomial of degree two $p_2(t) = x_1 + x_2 t + x_3 t^2$ which interpolates the three data points

t	y
-2	-27
0	-1
1	0

In the monomial basis, the coeffecients are the solution of the system of linear equations

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} 1 & t_1 & t_1^2 \\ 1 & t_2 & t_2^2 \\ 1 & t_3 & t_3^2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \mathbf{y}$$

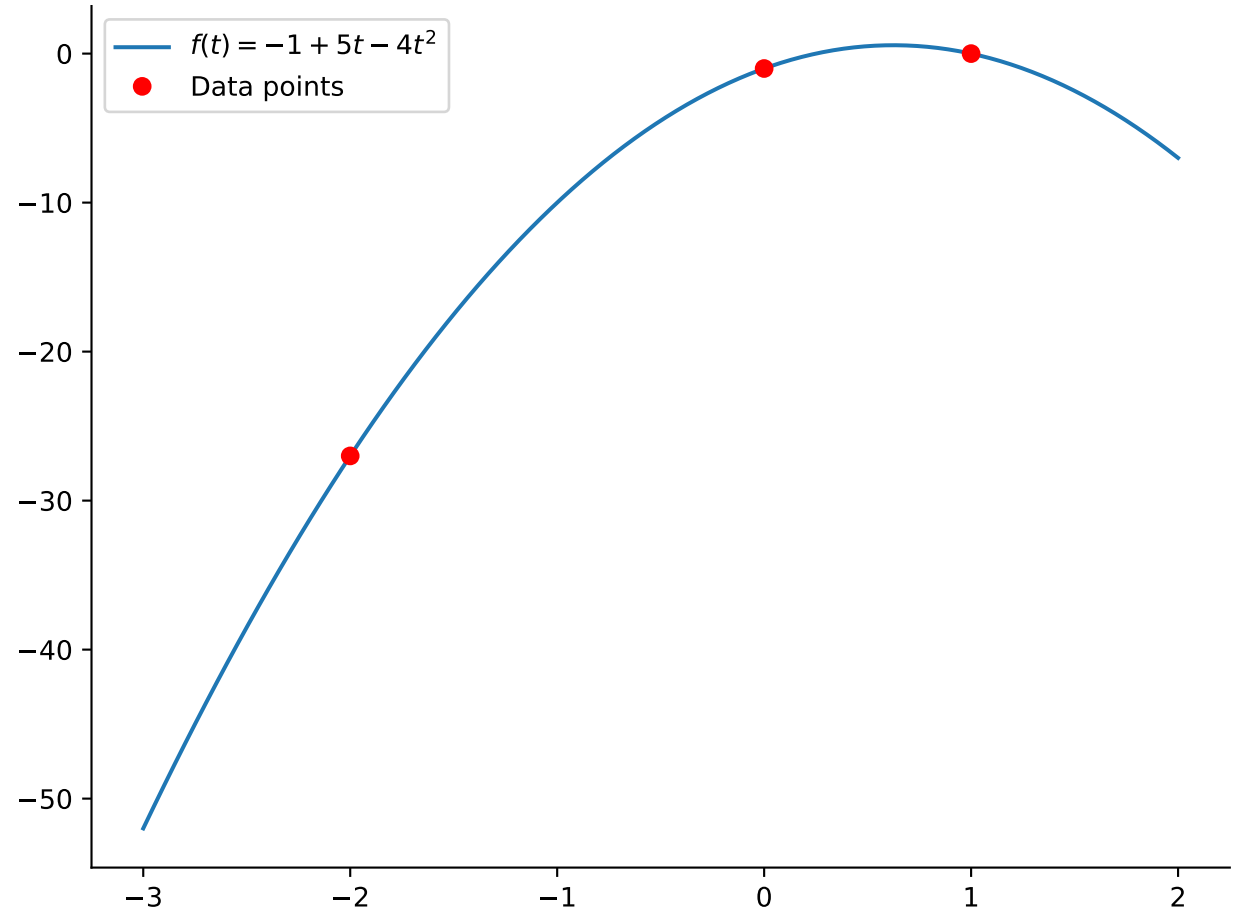
which becomes

$$\begin{bmatrix} 1 & -2 & 4 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -27 \\ -1 \\ 0 \end{bmatrix}$$

Solving this system using the methods seen before, we find the solution $\mathbf{x}_\intercal = [-1, 5, -4]$, so that the interpolating polynomial is

$$p_2(t) = -1 + 5t - 4t^2$$

```
def demo_mono_interpol():
    """Plot interpolation of data set using a monomial basis"""

    def f(t):
        """Define interpolating polynomial"""
        return -1 + 5 * t - 4 * t**2

    plt.close("mono")
    fig, ax = plt.subplots(num="mono")
    points = [(-2, -27), (0, -1), (1, 0)]
    x_points, y_points = zip(*points, strict=False)
    t_values = np.linspace(-3, 2, 100)
    ax.plot(t_values, f(t_values), label="$f(t) = -1 + 5t - 4t^2$")
    ax.plot(x_points, y_points, "ro", label="Data points")
    ax.legend()


demo_mono_interpol()
```



Such a matrix, whose columns are subsequent powers of a variable is called a **Vandermonde matrix**.

Although it is not singular, it is often *nearly singular* because the functions become increasingly difficult to distinguish as the degrees increase (see figure below). This makes the columns of the Vandermonde matrix almost linearly dependent.

```
# Helper functions to make nice inline labels on plots.
# Adapted from https://stackoverflow.com/questions/16992038/inline-labels-in-
↪matplotlib


def label_line(line, x, label=None, **kwargs):
    """Label line with line2D label data."""
    ax = line.axes
    xdata = line.get_xdata()
    ydata = line.get_ydata()

    if (x < xdata[0]) or (x > xdata[-1]):
        print("x label location is outside data range!")
        return

    # Find corresponding y co-ordinate and angle of the line
    ip = 1
    for i in range(len(xdata)):
        if x < xdata[i]:
            ip = i
            break
```

(continues on next page)

(continued from previous page)

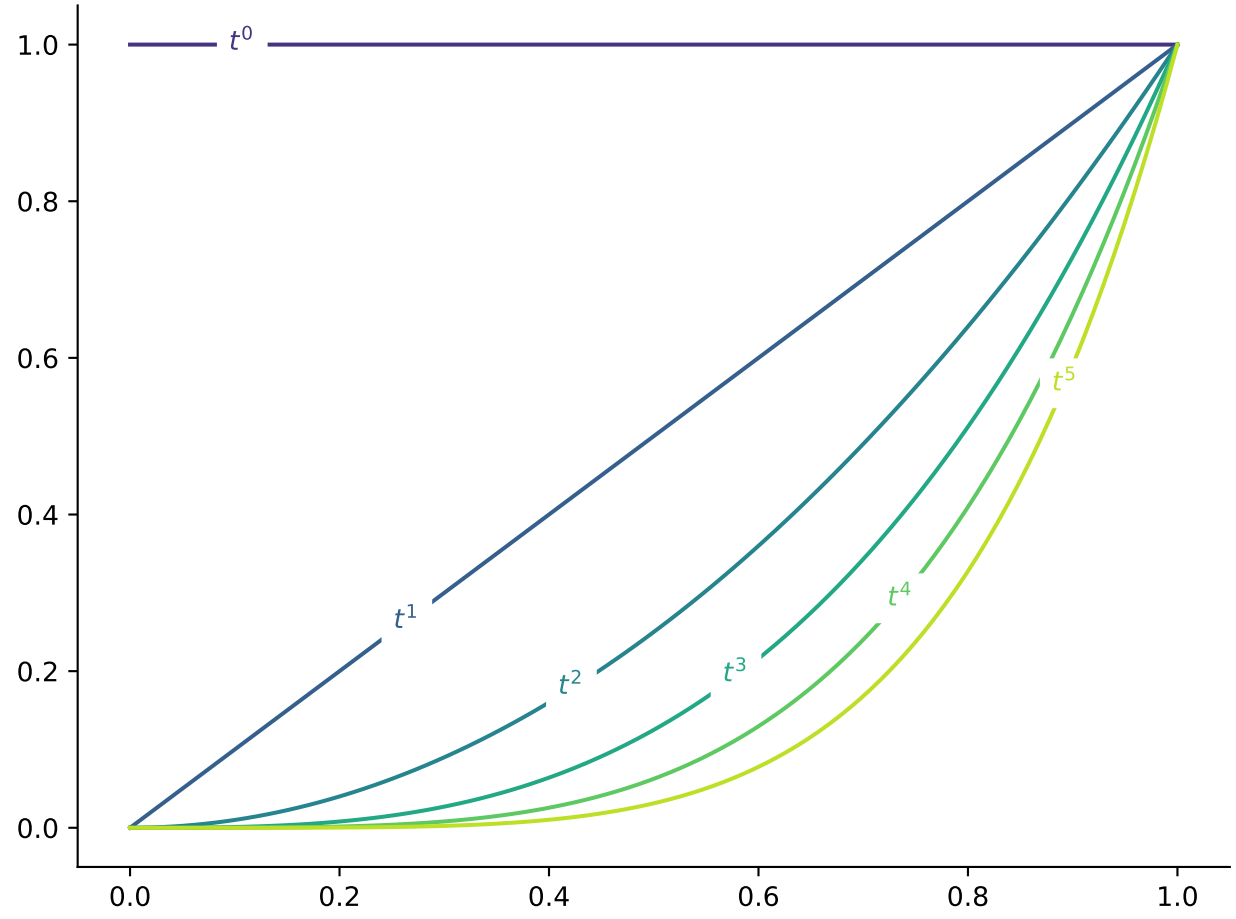
```
    y = ydata[ip - 1] + (ydata[ip] - ydata[ip - 1]) * (x - xdata[ip - 1]) / (
        xdata[ip] - xdata[ip - 1]
    )

    if not label:
        label = line.get_label()

    # Set a bunch of keyword arguments
    if "color" not in kwargs:
        kwargs["color"] = line.get_color()

    if ("horizontalalignment" not in kwargs) and ("ha" not in kwargs):
        kwargs["ha"] = "center"

    if ("verticalalignment" not in kwargs) and ("va" not in kwargs):
        kwargs["va"] = "center"

    if "backgroundcolor" not in kwargs:
        kwargs["backgroundcolor"] = ax.get_facecolor()

    if "clip_on" not in kwargs:
        kwargs["clip_on"] = True

    if "zorder" not in kwargs:
        kwargs["zorder"] = 2.5

    ax.text(x, y, label, **kwargs)


def label_lines(ax, xvals=None, **kwargs):
    lines = []
    labels = []

    # Take only the lines which have labels other than the default ones
    for line in ax.get_lines():
        label = line.get_label()
        if "_line" not in label:
            lines.append(line)
            labels.append(label)

    if xvals is None:
        xmin, xmax = ax.get_xlim()
        xvals = np.linspace(xmin, xmax, len(lines) + 2)[1:-1]

    for line, x, label in zip(lines, xvals, labels, strict=False):
        label_line(line, x, label, **kwargs)
```

```
def monomial_basis(n):
    """Plot the first n monomial basis functions.

    Parameters
    ----------
    n
```

(continues on next page)

(continued from previous page)

```
        The number of basis functions.

    """
    F = np.zeros((100, n + 1))
    F[:, 0] = np.linspace(0, 1, 100)
    plt.close("mono2")
    fig, ax = plt.subplots(num="mono2")
    for i in range(1, n + 1):
        F[:, i] = F[:, 0] ** (i - 1)
        color = plt.cm.viridis(0.75 * i / (n - 1))
        ax.plot(F[:, 0], F[:, i], color=color, label=f"$t^{i-1!s}$")

    label_lines(ax)


monomial_basis(6)
```



Next to the computational cost of determining the interpolating polynomial, the cost of evaluating it is also an important factor in choosing the appropriate method.

When represented in the monomial basis, a polynomial

$$p_{n-1}(t) = x_1 + x_2 t + x_3 t^2 + \cdots + x_n t^{n-1}$$

can be evaluated very efficiently using **Horner's method** also known as **Nested evaluation** or **synthetic division**:

$$p_{n-1}(t) = x_1 + t(x_2 + t(x_3 + t(\cdots(x_{n-1} + x_n t)\cdots)))$$

which requires only n summations and n additions.

7.2.2 Lagrange interpolation

For a given set of data points $(t_i, y_i), i = 1, \ldots, m$, the **Langrange basis functions** for $\mathbb{P}_{n-1}$ are given by

$$l_j(t) = \frac{\prod_{k=1,k\neq j}^{n}(t - t_k)}{\prod_{k=1,k\neq j}^{n}(t_j - t_k)}$$

It can be seen that:

- $l_j(t)$ is a polynomial of degree $n - 1$.
- At the edges of each interval, we have

$$l_j(t_i) = \begin{cases} 1, & \text{if} \quad i = j. \\ 0, & \text{if} \quad i \neq j. \end{cases}$$

  which means that for this basis the matrix of the linear system $\mathbf{Ax} = \mathbf{y}$ is the identity matrix $\mathbf{I}$.

The interpolating polynomial then is

$$p_{n-1}(t) = y_1 l_1(t) + y_2 l_2(t) + \cdots + y_n l_n(t)$$

which is easy to construct.

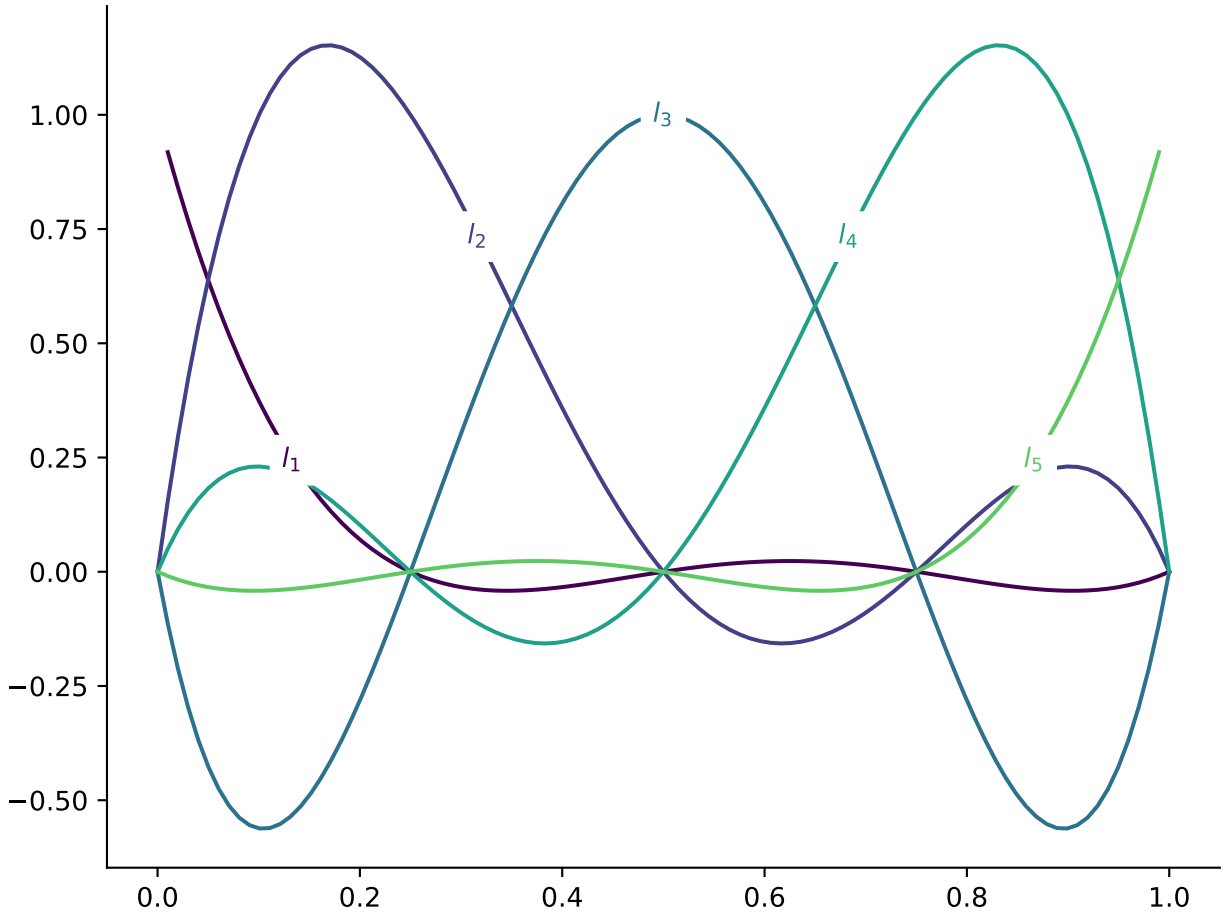
The figure below shows the Lagrange basis functions for n equally spaced points on the interval $[0, 1]$.

```
def lagrange_basis(n):
    """Plot the first n lagrange basis functions.

    Parameters
    ----------
    n
        The number of basis functions.

    """
    t = np.linspace(0, 1, 100)
    T = np.linspace(0, 1, n)
    L = np.ones(100)
    W = np.ones(n)
    for i in range(n):
        L = L * (t - T[i])

    for i in range(n):
        for j in range(n):
            if i != j:
                W[i] = W[i] / (T[i] - T[j])

    l = np.ones((n, 100))
    for i in range(0, n):
        l[i] = L * W[i] / (t - T[i])

    plt.close("lagrange")
```

(continues on next page)

(continued from previous page)

```
    fig, ax = plt.subplots(num="lagrange")
    for i in range(n):
        color = plt.cm.viridis(0.75 * i / (n - 1))
        ax.plot(t, l[i], color=color, label=f"$l_{i+1!s}$")

    label_lines(ax)


lagrange_basis(5)
```



Example

For the same data points as the previous example

t	y
-2	-27
0	-1
1	0

we find

$$l_1(t) = \frac{(t-0)(t-1)}{(-2-0)(-2-1)} = \frac{t(t-1)}{6}$$
$$l_2(t) = \frac{(t+2)(t-1)}{(0+2)(0-1)} = \frac{(t+2)(t-1)}{-2}$$
$$l_3(t) = \frac{(t+2)(t-0)}{(1+2)(1-0)} = \frac{(t+2)t}{3}$$

$$p_2(t) = \sum_{i=1}^{3} y_i l_i(t) = \frac{-27}{6}t(t-1) + \frac{1}{2}(t+2)(t-1) + \frac{0}{3}t(t+2)$$

which simplifies to

$$p_2(t) = -1 + 5t - 4t^2$$

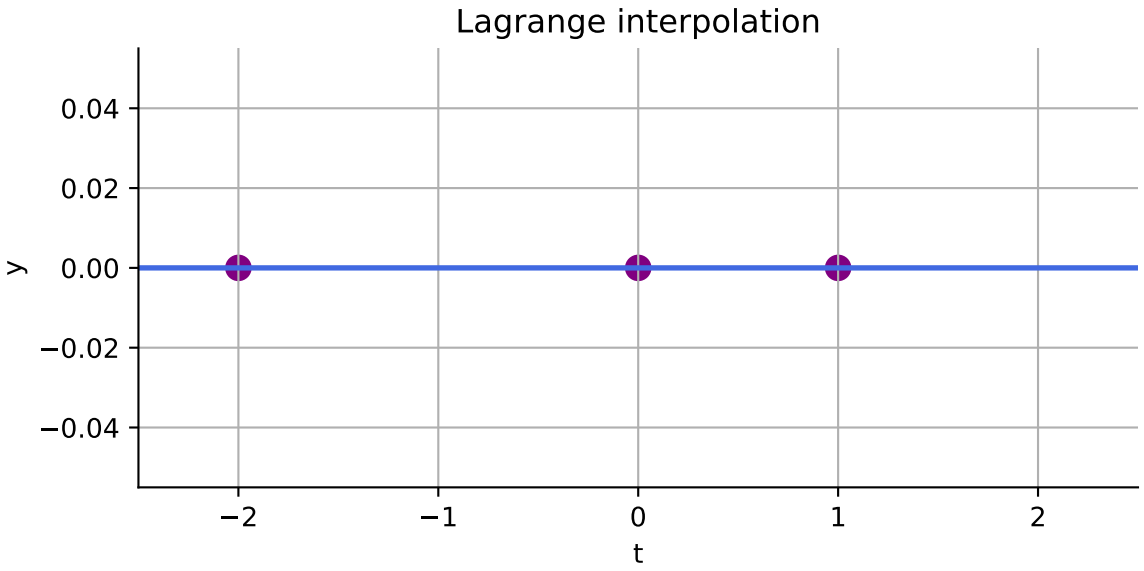
```
# The same example, now with scipy
interpolate.lagrange(np.array([-2, 0, 1]), np.array([-27, -1, 0]))
```

```
poly1d([-4.,  5., -1.])
```

```
def plot_lagrange(y1, y2, y3):
    poly_coef = interpolate.lagrange(np.array([-2, 0, 1]), (y1, y2, y3))

    tt = np.linspace(-3, 3, 100)
    yy = poly_coef[2] * tt**2 + poly_coef[1] * tt + poly_coef[0]

    fig, ax = plt.subplots(num="lagrange2", clear=True, figsize=(6, 3))
    ax.scatter(np.array([-2, 0, 1]), (y1, y2, y3), s=80, c="purple")
    ax.plot(tt, yy, c="royalblue", linewidth=2)
    ax.grid()
    ax.set_xlim(-2.5, 2.5)
    ax.set_xlabel("t")
    ax.set_ylabel("y")
    ax.set_title("Lagrange interpolation")


plt.close("lagrange2")
widgets.interact(plot_lagrange, y1=(-30, 30, 1), y2=(-30, 30, 1), y3=(-30, 30,
 ↪ 1))
```

```
interactive(children=(IntSlider(value=0, description='y1', max=30, min=-30),␣
 ↪IntSlider(value=0, description='y…
```

```
<function __main__.plot_lagrange(y1, y2, y3)>
```



7.2.3 Newton interpolation

In the previous two cases, we encountered a basis matrix **A** which is full (monomial) or diagonal (Lagrange). For **Newton interpolation** the basis matrix will be triangular.

For a given set of data points $(t_i, y_i), i = 1, \ldots, m$, the **Newton basis functions** for $\mathbb{P}_{n-1}$ are given by

$$\pi_j(t) = \prod_{k=1}^{j-1}(t - t_k)$$

Note that we assign $\pi_j(t) = 1$ for $j = 1$

The interpolating polynomial then has the form

$$p_{n-1}(t) = x_1 + x_2(t - t_1) + x_3(t - t_1)(t - t_2) + \cdots x_n(t - t_1)\ldots(t - t_{n-1})$$

From the definition it can be seen that the basis matrix **A** is lower triangular, so the system $\mathbf{Ax} = \mathbf{y}$ can efficiently be solved by forward substitution.

Furthermore, once we know the coefficients x_i, the resulting polynomial can efficiently be evaluated using Horner's nested scheme

$$p_{n-1}(t) = x_1 + (t - t_1)\left[x_2 + (t - t_2)\left[x_3 + (t - t_3)\left[\cdots(x_{n-1} + x_n(t - t_{n-1}))\cdots\right]\right]\right]$$

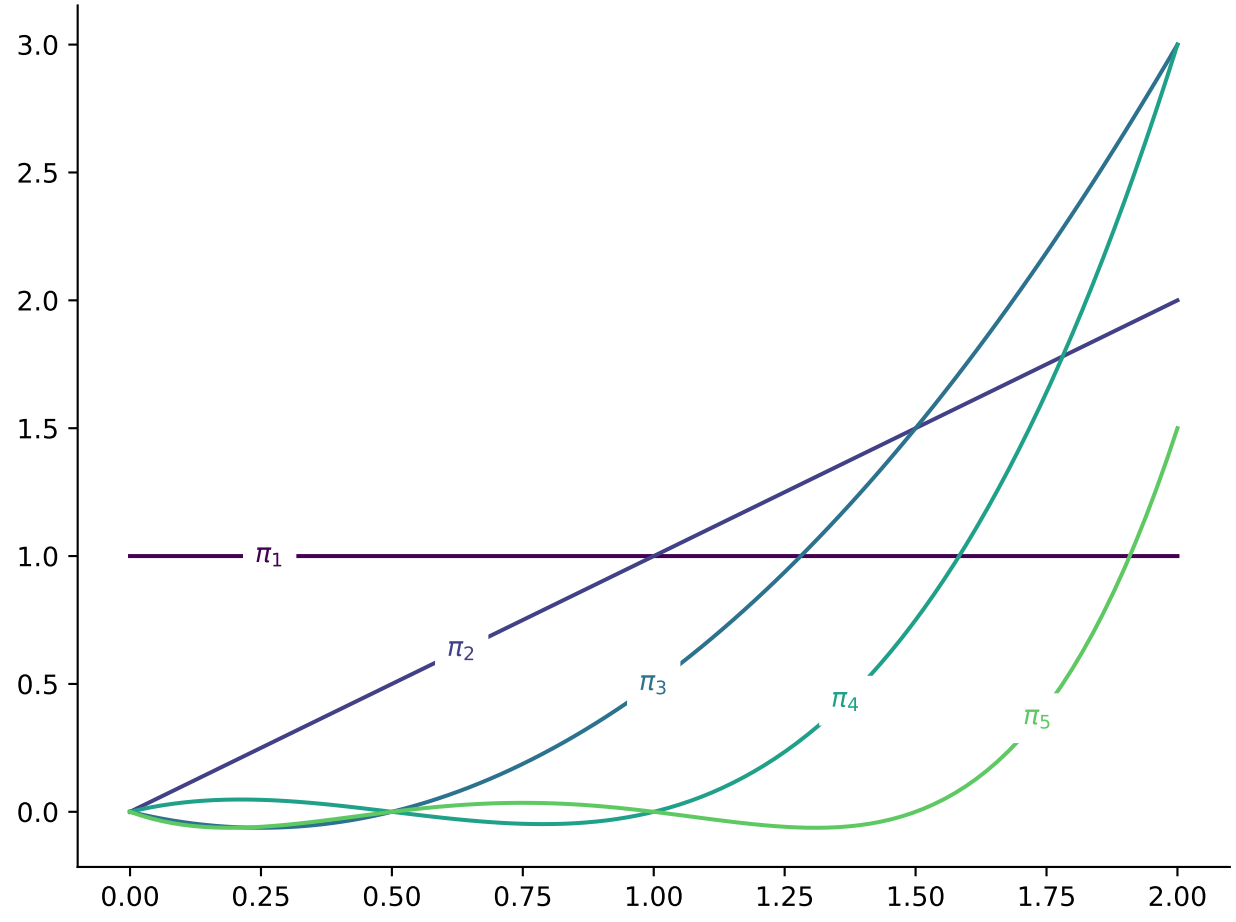
The cell below shows the newton basis functions for n equally spaced points on the interval $[0, 2]$.

```
def newton_basis(n):
    """Plot the first n newton basis functions.

    Parameters
    ----------
    n
        The number of basis functions.

    """
    t = np.linspace(0, 2, 100)
    T = np.linspace(0, 2, n)
```

(continues on next page)

(continued from previous page)

```
    P = np.ones((n, 100))

    for i in range(n):
        for j in range(i):
            P[i] = P[i] * (t - T[j])

    plt.close("newton")
    fig, ax = plt.subplots(num="newton")
    for i in range(n):
        color = plt.cm.viridis(0.75 * i / (n - 1))
        ax.plot(t, P[i], color=color, label=rf"$\pi_{i+1!s}$")

    label_lines(ax)


newton_basis(5)
```



Example

In general we have:

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & t_2 - t_1 & 0 \\ 1 & t_3 - t_1 & (t_3 - t_1)(t_3 - t_2) \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

filling in the same 3 data points as before, we find

t	y
-2	-27
0	-1
1	0

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 2 & 0 \\ 1 & 3 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -27 \\ -1 \\ 0 \end{bmatrix}$$

Using forward substitution, we find $\mathbf{x}^\intercal = [-27, 13, 4]$, so the interpolating polynomial is

$$p(t) = -27 + 13(t + 2) - 4(t + 2)t$$

which (as expected) simplifies to

$$p(t) = -1 + 5t - 4t^2$$

Another useful property of the newton basis functions is that the interpolant can be constructed *incrementally* as more data points are added.

If $p_j(t)$ is a polynomial of degree $j - 1$ which interpolates j data points, then for any constant x_{j+1}

$$p_{j+1}(t) = p_j(t) + x_{j+1}\pi_{j+1}(t)$$

is a polynomial of degree j that also interpolates the same j points. The free parameter x_{j+1} can be chosen so that $p_{j+1}(t)$ interpolates the new data points y_{j+1} as

$$x_{j+1} = \frac{y_{j+1} - p_j(t_{j+1})}{\pi_{j+1}(t_{j+1})}$$

Example

Starting from the previous example, we have

$$\begin{bmatrix} 1 & 0 \\ 1 & t_2 - t_1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

filling in first 2 data points, we find

$$\begin{bmatrix} 1 & 0 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -27 \\ -1 \end{bmatrix}$$

Using forward substitution, we find $\mathbf{x}_\intercal = [-27, 13]$, so the interpolating polynomial is

$$p_2(t) = -27 + 13(t + 2)$$

If we now want to add the additional data point [1,0], we can determine x_3 as

$$x_3 = \frac{y_3 - p_2(t_3)}{\pi_3(t_3)} = \frac{0 - [-27 + 13(1+2)]}{(1-0)(1+2)} = -4$$

such that

$$p_3(t) = p_2(t) + x_3\pi_3(t) = -27 + 13(t+2) - 4(t-0)(t+2) = -27 + 13(t+2) - 4(t+2)t$$

7.2.4 Polynomial interpolation of a continuous function

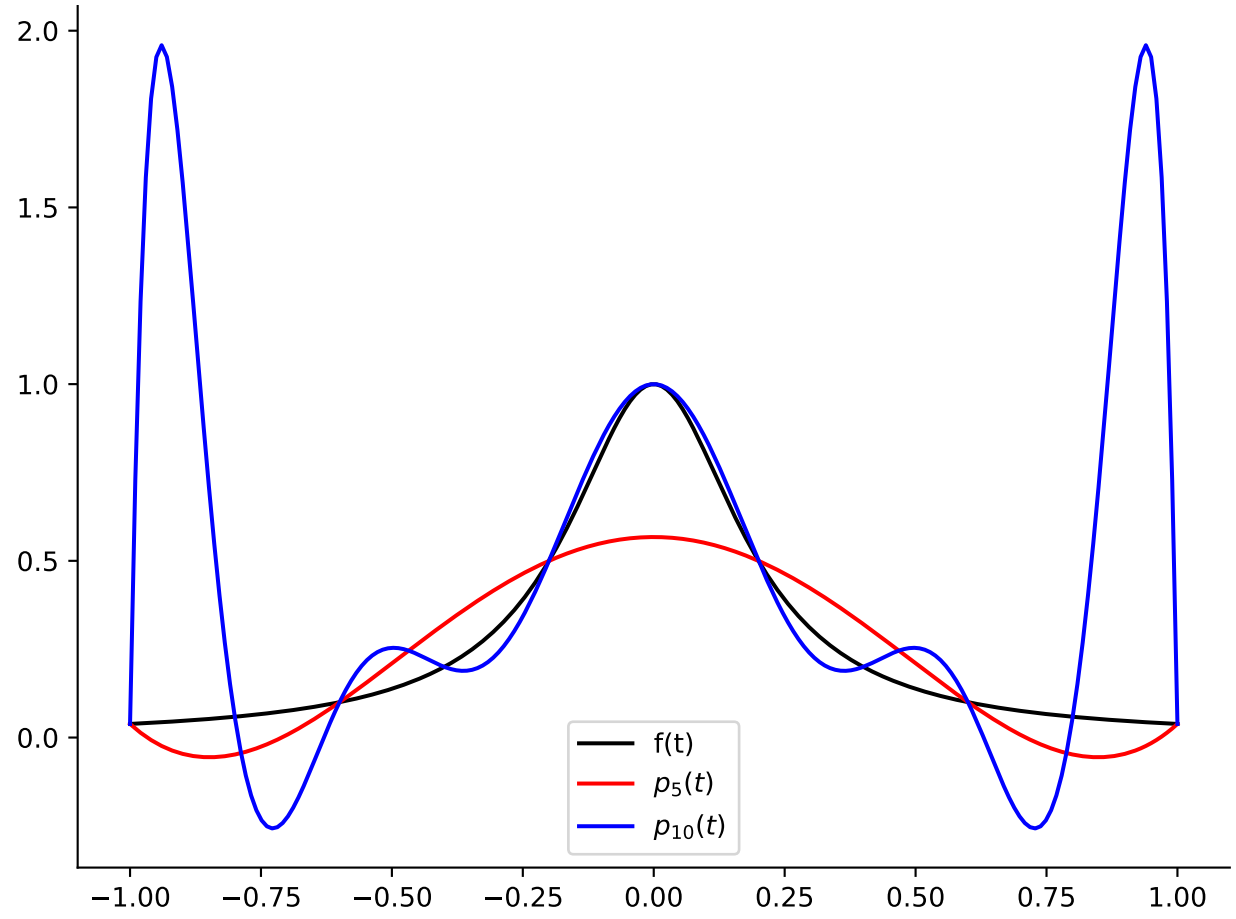
Interpolants of a high degree

- can be expensive to determine or evaluate (depending on the basis chosen)
- by definition a polynomial of degree n has $n-1$ extrema, and thus many "wiggles" Even if the polynomial passes through all required data points, it may fluctuate wildly in between these points and does not approximate an underlying function at all.

As an example, have a look at the Runge's function:

$$f(t) = \frac{1}{1 + 25t^2}$$

```
def runge(x):
    return 1.0 / (1 + 25 * x**2)


def demo_cont_interpolation():
    """A poor interpolation of a continuous function at equidistant points."""
    t = np.linspace(-1, 1, 200)
    # 6 equidistant points
    x6 = np.linspace(-1, 1, 6)
    # polynomial interpolant of degree 5
    poly5 = interpolate.lagrange(x6, runge(x6))

    # 11 equidsitant points
    x11 = np.linspace(-1, 1, 11)
    # polynomial interpolant of degree 10
    poly10 = interpolate.lagrange(x11, runge(x11))

    plt.close("runge")
    fig, ax = plt.subplots(num="runge")
    ax.plot(t, runge(t), "k", label="f(t)")
    ax.plot(t, poly5(t), "r", label=r"$p_5(t)$")
    ax.plot(t, poly10(t), "b", label=r"$p_{10}(t)$")
    ax.legend()


demo_cont_interpolation()
```



The example shown above learns us that polynomial interpolants of increasing degree converge to the function in the middle of the interval, but diverge near the endpoints. More satisfactory results can be obtained if our sample points are bunched near the ends of the interval.

One good way to achieve this is to use the **Chebysev points** which are defined on the interval $[-1, 1]$, but can transformed to an arbitrary interval.

$$t_i = \cos\left(\frac{(2i-1)\pi}{2k}\right), \qquad i = 1, \ldots, k$$

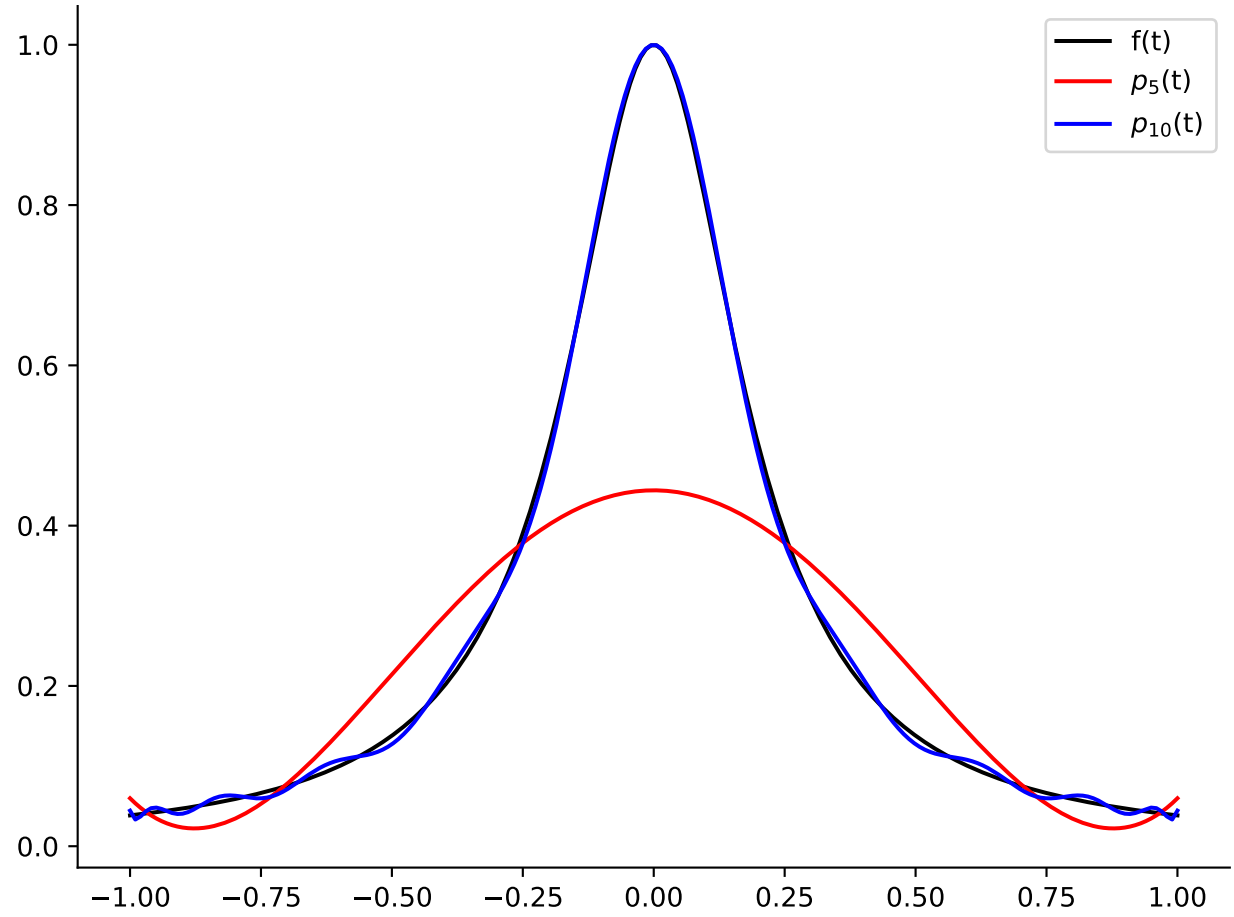
The example shown below shows that this indeed provides a better approximation to the underlying function.

```
def chebysev(n):
    """Return the 'n' chebysev points in the interval [0:1].

    Parameters
    ----------
    n
        The number of chebysev points.

    """
    cheb_points = np.zeros(n)
    for i in range(n):
        cheb_points[i] = np.cos((2.0 * (i + 1.0) - 1.0) * np.pi / (2.0 * n))
    return cheb_points
```

(continues on next page)

(continued from previous page)

```
def demo_cont_interpolation_chebysev():
    """Demo which illustrates that interpolating
    a continuous function evaluated at the chebysev
    points results in a better fit to the true function"""

    t = np.linspace(-1, 1, 200)
    # 6 chebysev points
    x6 = chebysev(6)
    # polynomial interpolant of degree 5
    poly5 = interpolate.lagrange(x6, runge(x6))

    # 11 chebysev points
    x21 = chebysev(21)
    # polynomial interpolant of degree 10
    poly10 = interpolate.lagrange(x21, runge(x21))

    plt.close("chebyshev")
    fig, ax = plt.subplots(num="chebyshev")
    ax.plot(t, runge(t), "k", label="f(t)")
    ax.plot(t, poly5(t), "r", label=r"$p_5$(t)")
    ax.plot(t, poly10(t), "b", label=r"$p_{10}$(t)")
    ax.legend()


demo_cont_interpolation_chebysev()
```



7.3 Piecewise polynomial interpolation

Fitting a a single polynomial to a large number of data points will most likely result in unsatisfactory oscillating behavior. An alternative is **piecewise polynomial interpolation** or **spline interpolation**: the interval is divided into subintervals, and a *different* polynomial of low degree is constructed in each subinterval. For this reason, the points at which the interpolant changes are called **knots**.

Consider a set of ordered data points (x_i, y_i) in the interval $[a, b]$. A function S is called a spline interpolation of degree k if

- S is defined on the interval $[a, b]$
- the rth derivative of S is continuous in the interval $[a, b]$ for $0 \leq r \leq k - 1$
- S is a polynomial of degree $\leq k$ in every subinterval between the knots

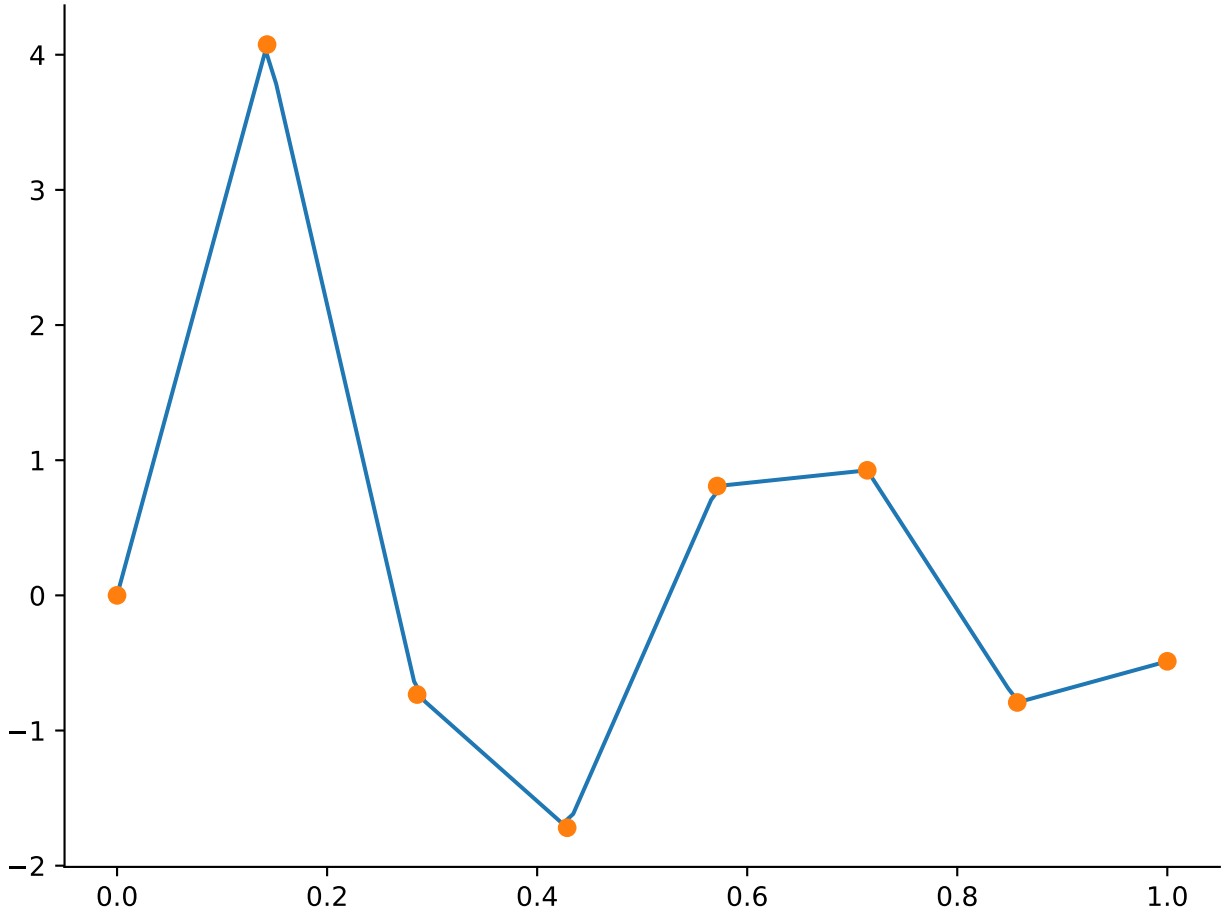
The simplest case is $k = 1$ for which S_j is a straight line and the spline is a piecewise linear interpolation, as shown below.

```
def linear_spline_demo():
    """Demo which shows a linear interplation between a few data points"""
    x = np.linspace(0, 1, 8)
    y = np.sin(12.0 * x) / (x + 0.1)
    f = interpolate.interp1d(x, y, kind="linear")
    t = np.linspace(0, 1, 100)
```

(continues on next page)

(continued from previous page)

```
    plt.close("pwlin")
    fig, ax = plt.subplots(num="pwlin")
    ax.plot(t, f(t), "-")
    ax.plot(x, y, "o")


linear_spline_demo()
```



7.3.1 Cubic spline interpolation

Theory

A spline with n knots has $(n-1)$ piecewise polynomials of degree k that interpolate the data. The number of free parameters thus is $(k+1)(n-1)$. For instance, a linear polynomial($k=1$, and with 2 free parameters per polynomial) has $2(n-1)$ free parameters and a **cubic spline** has $4(n-1)$.

- Interpolating the data requires $2(n-1)$ equations, because each of the $(n-1)$ polynomials must match the two data points at either end of the subinterval.
- Requiring the derivative to be continuous gives $(n-2)$ additional equations, because there are $(n-2)$ *interior* data points.
- Requiring that the second derivative is also continuous gives $(n-2)$ additional equations.

The spline of lowest degree that has sufficient variables to satisfy all these equations is a cubic spline ($4n-4$ variables for $4n-6$ conditions). The remaining two variables can be fixed in a number of ways:

- **clamped cubic spline**: Specifying the first derivative of the endpoints
- **natural spline**: forcing the second derivative to be zero at the endpoints
- **not-a-knot**: the third derivative of the spline is continuous at the one-but-outermost data points
- **periodic spline**: forcing equality of the first as well as second derivatives of the two outermost points (if the spline is to be periodic)

Example of `scipy.interpolate.CubicSpline` and its options

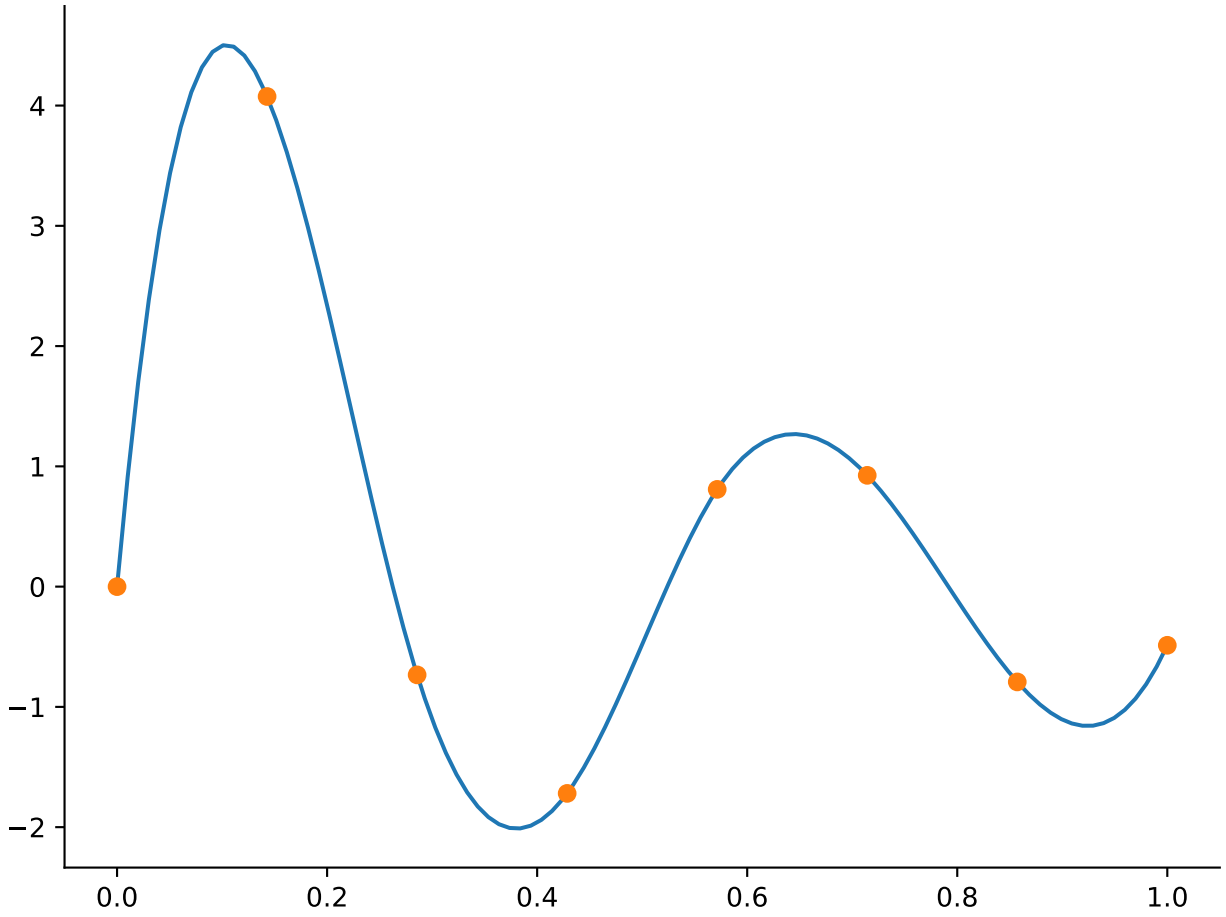
In the example below a cubic spline is shown through 10 data points sampled from a sine function. Play around with the `bc_type` option (all options shown below) to see how they affect the resulting spline.

The derivatives of the spline are also shown, so see for yourself that the first and second derivatives are indeed continuous.

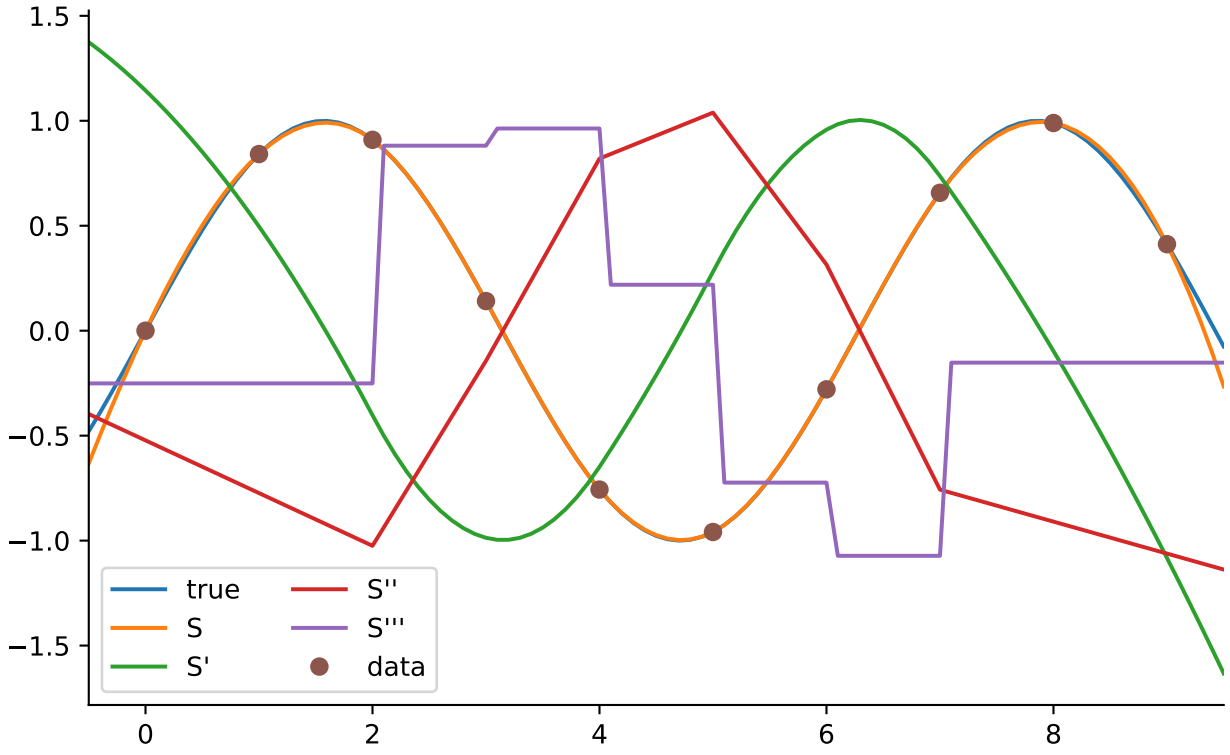
- `not-a-knot`: The first and second segment at a curve end are the same polynomial. It is a good default when there is no information on boundary conditions`
- `periodic`: The interpolated functions is assumed to be periodic of period `x[-1] - x[0]`. The first and last value of y must be identical: `y[0] == y[-1]`. This boundary condition will result in `y'[0] == y'[-1]` and `y''[0] == y''[-1]`.
- `clamped`: The first derivative at curves ends are zero. Assuming a 1D y, `bc_type=((1, 0.0), (1, 0.0))` is the same condition.
- `natural`: The second derivative at curve ends are zero. Assuming a 1D y, `bc_type=((2, 0.0), (2, 0.0))` is the same condition.

Source

```
def cubic_spline_demo():
    """Demo which shows a cubic spline interplation between a few data points"
 ↪""
    x = np.linspace(0, 1, 8)
    y = np.sin(12.0 * x) / (x + 0.1)
    f = interpolate.interp1d(x, y, kind="cubic")
    t = np.linspace(0, 1, 100)

    plt.close("cspline")
    fig, ax = plt.subplots(num="cspline")
    ax.plot(t, f(t), "-")
    ax.plot(x, y, "o")


cubic_spline_demo()
```




```
def cubic_spline_detailed_demo():
    """Demo which shows a cubic spline interplation between a few data points"
 ↪""
    x = np.arange(10)
    y = np.sin(x)
    cs = interpolate.CubicSpline(x, y, bc_type="not-a-knot")
    xs = np.arange(-0.5, 9.6, 0.1)

    plt.close("cspline2")
    fig, ax = plt.subplots(num="cspline2", figsize=(6.5, 4))
    ax.plot(xs, np.sin(xs), label="true")
    ax.plot(xs, cs(xs), label="S")
    ax.plot(xs, cs(xs, 1), label="S'")
    ax.plot(xs, cs(xs, 2), label="S''")
    ax.plot(xs, cs(xs, 3), label="S'''")
    ax.plot(x, y, "o", label="data")
    ax.set_xlim(-0.5, 9.5)
    ax.legend(loc="lower left", ncol=2)


cubic_spline_detailed_demo()
```



Example

Let's say that we want to fit a **clamped cubic spline** to the following points

x	y
0	0
1	2
3	0

This gives us 12 equations in total for the two splines

$$S_1 = a_1x^3 + b_1x^2 + c_1x + d_1$$

and

$$S_2 = a_2x^3 + b_2x^2 + c_2x + d_2$$

with their respective first and second derivatives:

$$S = \begin{cases} S_1(0) = d_1 = 0 & \text{I} \\ S_1(1) = a_1 + b_1 + c_1 = 2 & \text{II} \\ S_2(1) = a_2 + b_2 + c_2 + d_2 = 2 & \text{III} \\ S_2(3) = 27a_2 + 9b_2 + 3c_2 + d_2 = 0 & \text{IV} \end{cases}$$

$$S' = \begin{cases} S_1'(0) = c_1 = 0 & \text{V} \\ S_1'(1) = 3a_1 + 2b_1 & \text{VIa} \\ S_2'(1) = 3a_2 + 2b_2 + c_2 & \text{VIb} \\ S_2'(3) = 27a_2 + 6b_2 + c_2 = 0 & \text{VII} \end{cases}$$

$$S'' = \begin{cases} S_1''(0) = 2b_1 & \\ S_1''(1) = 6a_1 + 2b_1 & \text{VIIIa} \\ S_2''(1) = 6a_2 + 2b_2 & \text{VIIIb} \\ S_2''(3) = 18a_2 + 2b_2 & \end{cases}$$

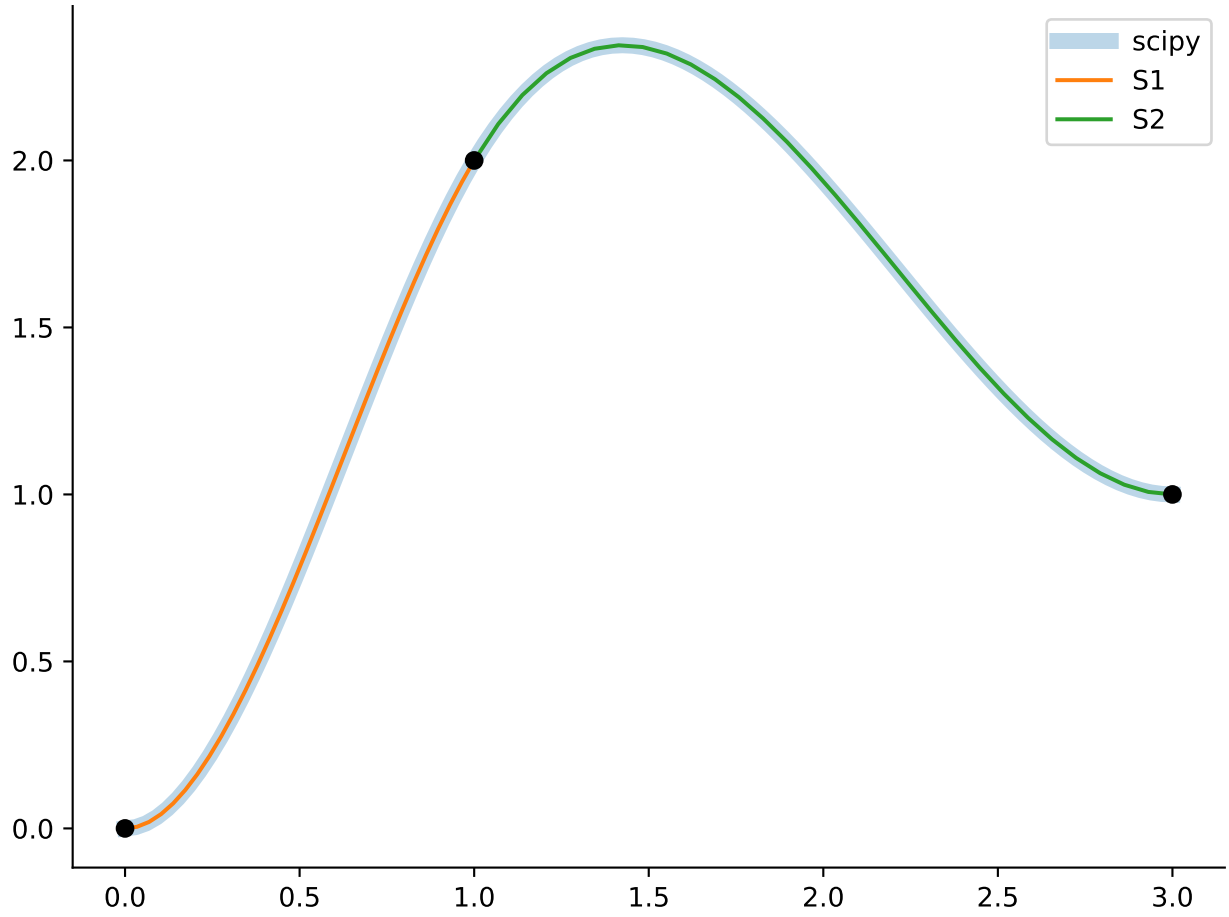
This results in a system of 6 linear equations (II, III, IV, VIa - VIb, VII and VIIIa - VIIIb should all equal zero) for the 6 unknowns $[a_1, b_1, a_2, b_2, c_2, d_2]$ (because c_1 and d_1 are 0). It can be solved with LU-factorisation as follows:

```
def interpolation_demo():
    # x = np.array([0, 1, 3, 4])
    # y = np.array([0, 2, 1, 0])
    b = np.array(
        [
            2,  # II
            2,  # III
            1,  # IV
            0,  # VIa - VIb
            0,  # VII
            0,  # VIIIa - VIIIb
        ]
    )
    A = np.array(
        [
            [1, 1, 0, 0, 0, 0],  # II
            [0, 0, 1, 1, 1, 1],  # III
            [0, 0, 27, 9, 3, 1],  # IV
            [3, 2, -3, -2, -1, 0],  # VIa - VIb
            [0, 0, 27, 6, 1, 0],  # VII
            [6, 2, -6, -2, 0, 0],  # VIIIa - VIIIb
        ]
    )
    coeff = linalg.lu_solve(linalg.lu_factor(A), b)
    return coeff


def s1(x):
    return a1 * x**3 + b1 * x**2


def s2(x):
    return a2 * x**3 + b2 * x**2 + c2 * x + d2


a1, b1, a2, b2, c2, d2 = interpolation_demo()
print(a1, b1, a2, b2, c2, d2)
```

```
-2.25 4.25 0.6875 -4.5625 8.8125 -2.9375
```

> We can now plot both functions to compare them to each other

```
def plot_example():
    plt.close("example")
    fig, ax = plt.subplots(num="example")

    # Calculate our splines using the interpolate.cubicsplines method
    f = interpolate.CubicSpline(
        np.array([0, 1, 3]), np.array([0, 2, 1]), bc_type="clamped"
    )
    # arange points along the x-axis to plot our function on
    xnew = np.linspace(0, 3, 100)
    ynew = f(xnew)
    ax.plot(xnew, ynew, lw=6, alpha=0.3, label="scipy")

    # plot S1
    x1 = np.linspace(0, 1, 30)
    ax.plot(x1, s1(x1), label="S1")

    # plot S2
    x2 = np.linspace(1, 3, 30)
    ax.plot(x2, s2(x2), label="S2")

    # plot the points that we interpolated through
    ax.plot(np.array([0, 1, 3]), np.array([0, 2, 1]), "ko")
    ax.legend()


plot_example()
```



> This shows that both methods return the same function. In practice you can always use the interpolate method that's already built in python to avoid any tedious work.

7.4 Software

`scipy.interpolate` link contains a lot of different functions, many of which we have not considered here.

The main functions corresponding to the theory covered here are

- `lagrange` for Lagrange interpolation
- `CubicSpline` for cubic spline interpolation (with many options)
- `interp1d` to interpolate 1 dimensional data (also capable of finding cubic splines, but with less options)
- `griddata` to interpolate multidimensional data

7.4.1 `interp1d`

Documentation

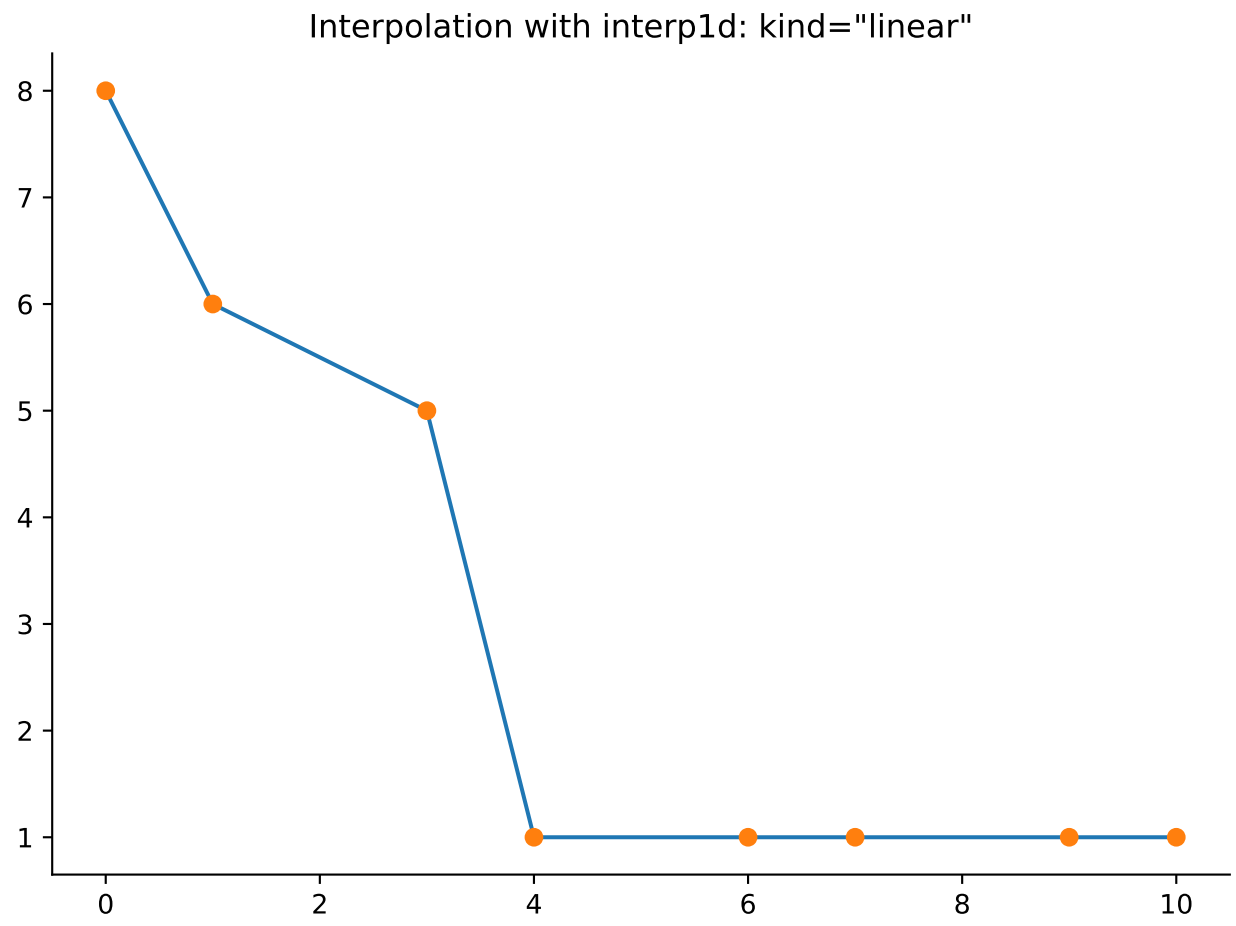
In the example below, you can change `kind` to see the different possible interpolations.

> **Warning**: interp1d is considered legacy API and is not recommended for use in new code. Consider using more

> specific interpolators instead. It is still used in this notebook because it allows for a straightforward comparison of different interpolations.

```
def interp1d_demo(kind):
    """Demo to show the different kinds of 1D interpolation.

    Parameters
    ----------
    kind
        The kind of interpolation:
        "linear", "nearest", "zero", "slinear", "quadratic",
        "cubic", "previous", or "next".
    """

    x = np.array([0, 1, 3, 4, 6, 7, 9, 10])
    y = np.array([8, 6, 5, 1, 1, 1, 1, 1])
    f = interpolate.interp1d(x, y, kind=kind)
    xnew = np.arange(0, 10, 0.1)
    ynew = f(xnew)

    fig, ax = plt.subplots(num="interp1d", clear=True)
    ax.plot(xnew, ynew, "-")
    ax.plot(x, y, "o")
    ax.set_title(f'Interpolation with interp1d: kind="{kind}"')


plt.close("interp1d")
widgets.interactive(
    interp1d_demo,
    kind=widgets.Dropdown(
        options=[
            "linear",
            "nearest",
            "zero",
            "slinear",
            "quadratic",
            "cubic",
            "previous",
            "next",
        ],
        description="Interpolation:",
    ),
)
```

```
interactive(children=(Dropdown(description='Interpolation:', options=('linear
↪', 'nearest', 'zero', 'slinear', …
```



7.4.2 griddata

Documentation

Suppose you have multidimensional data, for instance, for an underlying function $f(x, y)$ you only know the values at points $(x[i], y[i])$ that do not form a regular grid.

Suppose we want to interpolate the 2-D function

$$f(x, y) = x(1 - x)\cos(4\pi x)\sin(4\pi y^2)^2$$

on a grid in $[0, 1] \times [0, 1]$ but we only know its values at n data points.

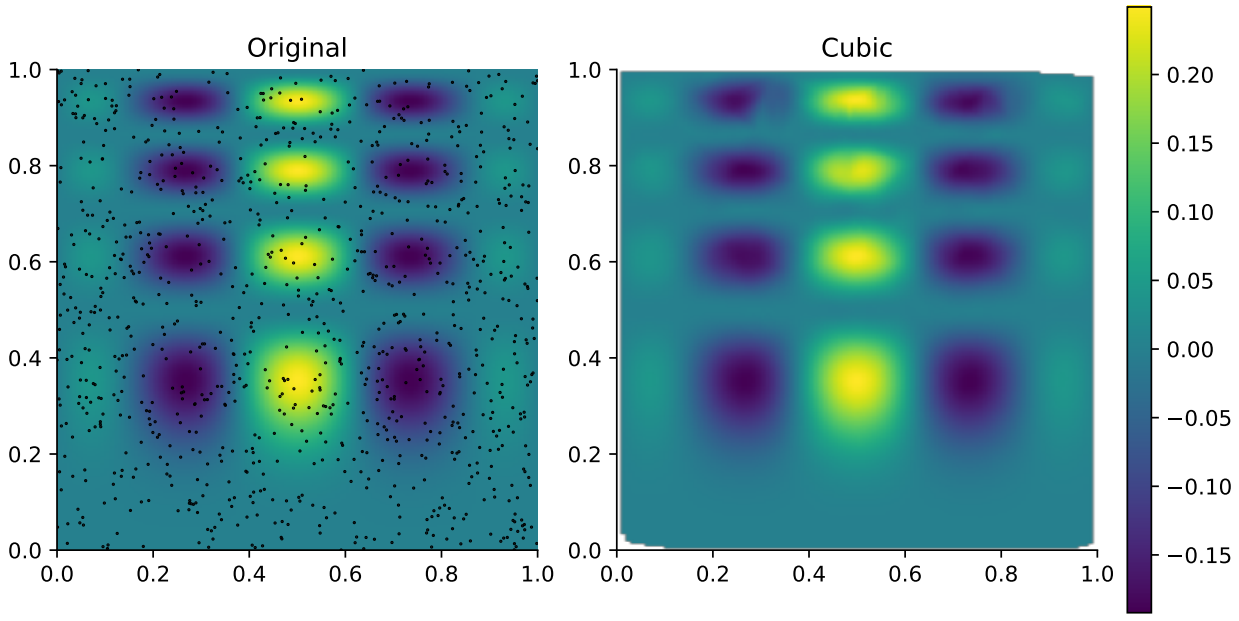
Then we can use `griddata` to interpolate our function

```
def griddata_demo(n):
    """Demo to show the usage of the griddata function.

    Parameters
    ----------
    n        The number of data points we're interpolating between
    """

    def func(x, y):
        return x * (1 - x) * np.cos(4 * np.pi * x) * np.sin(4 * np.pi * y**2)␣
↪** 2
```

(continues on next page)

(continued from previous page)

```
    grid_x, grid_y = np.mgrid[0:1:100j, 0:1:200j]
    rng = np.random.default_rng()
    points = rng.random((n, 2))
    values = func(points[:, 0], points[:, 1])
    grid_z2 = interpolate.griddata(points, values, (grid_x, grid_y), method=
↪"cubic")
    plt.close("griddata")
    fig, axs = plt.subplots(
        1,
        3,
        figsize=(8, 4),
        gridspec_kw={"width_ratios": [1, 1, 0.05]},
        num="griddata",
    )
    axs[0].imshow(func(grid_x, grid_y).T, extent=(0, 1, 0, 1), origin="lower")
    axs[0].plot(points[:, 0], points[:, 1], "k.", ms=1)
    axs[0].set_title("Original")
    img = axs[1].imshow(grid_z2.T, extent=(0, 1, 0, 1), origin="lower")
    axs[1].set_title("Cubic")
    fig.colorbar(img, cax=axs[2])


griddata_demo(1000)
```



CHAPTER 8

Numerical Integration and Differentiation

```
import timeit

import matplotlib.pyplot as plt
import numpy as np
from ipywidgets import interact, widgets
from matplotlib.patches import Rectangle
from scipy import integrate, optimize
```

8.1 Integration

In elementary geometry you learned how to calculate the area and volume of simple shapes like circles, cubes, etc…

The topic of this notebook (in contrast to the notebooks on solving ODE's and PDE's, where we will see methods to solve differential equations) is to determine the area under a curve and numerically differentiating curves.

In fact, one of the motivations for the invention of integral calculus was the need to also calculate the area of more complex, irregular shapes.

Methods for approximating the area of such irregular shapes were already known by Archimedes, whose approach was to tile the region with small squares (for which he know the area) and then count the total number of squares.

The principle behind this method is not too far from the currently used methods.

Theoretically, for a function $f : \mathbb{R} \to \mathbb{R}$ on an interval $[a, b]$, the definition of the integral

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

is based on the **Riemann sums** of the form

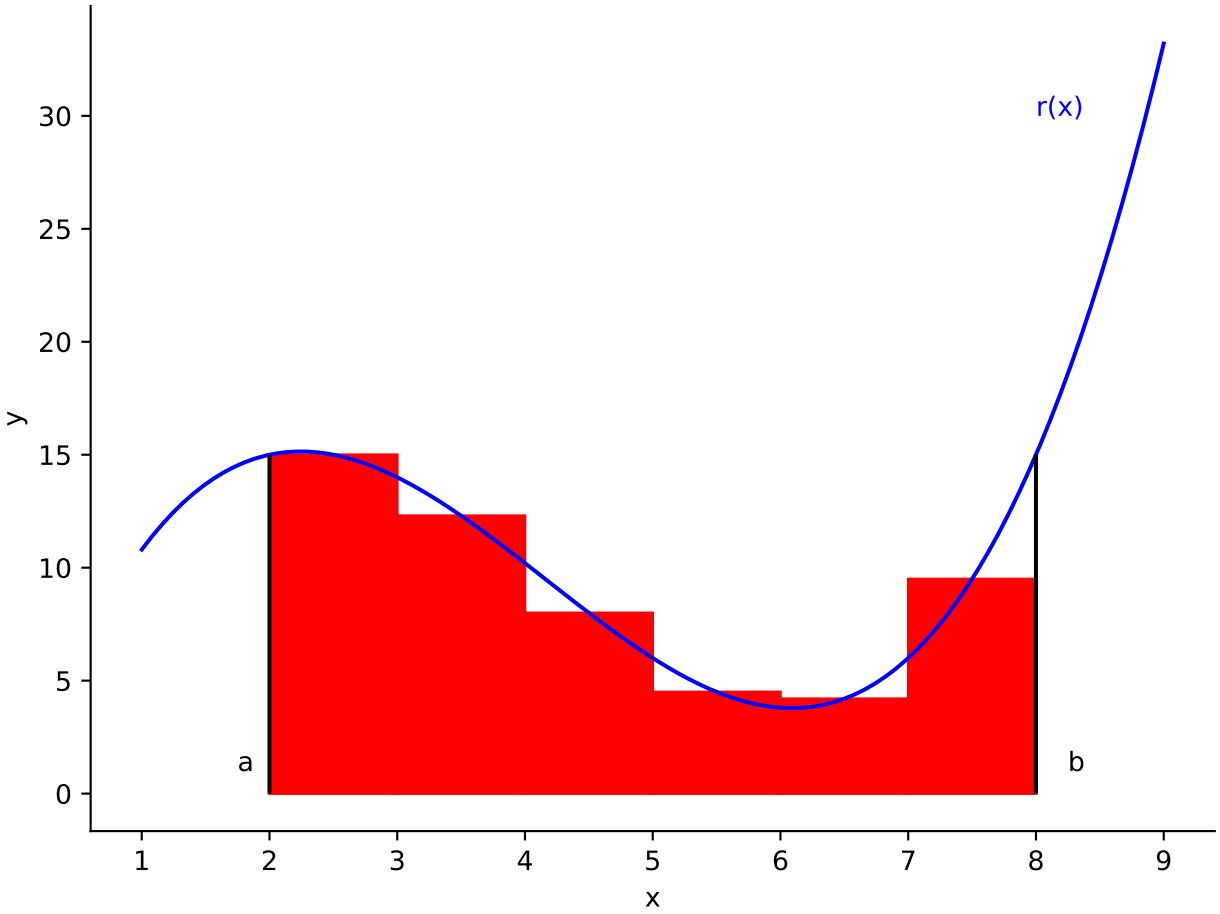
$$R_n = \sum_{i=1}^{n} (x_{i+1} - x_i) f(\xi_i)$$

where $a = x_1 < x_2 < \cdots < x_n < x_{n+1} = b$ and $\xi_i \in [x_i, x_{i+1}]$, $i = 1, \ldots, n$.

Let $h_n = \max\{x_{i+1} - x_i : i = 1, \ldots, n-1\}$. If for any choice of x_i such that $h_n \to 0$ and any choice of ξ_i, we have $\lim_{n\to\infty} R_n = R$, where R is finite, then f is said to be **Riemann integrable** on $[a, b]$, and the value of the integral is R.

This method already suggests a method for approximating an integral: just use a finite Riemann sum with n chosen large enough to achieve the desired accuracy. This idea works in principle, but in practice will be far from optimal in comparison to more efficient choices which carefully select x_i and ξ_i.

```
def plot_concept():
    def integrand(x):
        return 1 / 15 * (6 * x - 3) * (x - 5) * (x - 7) + 6

    plt.close("concept")
    fig, ax = plt.subplots(num="concept")
    ts = np.linspace(1, 9, 100)
    ax.plot(ts, integrand(ts), color="Blue")
    ax.vlines(x=2, ymin=0, ymax=integrand(2), color="black")
    ax.vlines(x=8, ymin=0, ymax=integrand(8), color="black")
    rectangles = [
        Rectangle((2, 0), 1, integrand(2.5), color="r"),
        Rectangle((3, 0), 1, integrand(3.5), color="r"),
        Rectangle((4, 0), 1, integrand(4.5), color="r"),
        Rectangle((5, 0), 1, integrand(5.5), color="r"),
        Rectangle((6, 0), 1, integrand(6.5), color="r"),
        Rectangle((7, 0), 1, integrand(7.5), color="r"),
    ]
    for rectangle in rectangles:
        ax.add_patch(rectangle)
    ax.text(1.75, 1, "a")
    ax.text(8.25, 1, "b")
    ax.text(8, 30, "r(x)", color="Blue")
    ax.set_xlabel("x")
    ax.set_ylabel("y")


plot_concept()
```



8.2 Existence, Uniqueness, Conditioning

- For all practical purposes, a function is integrable if it is bounded (no singularities) with at most a finite number of points of discontinuity within the interval of integration.
- Since all the Riemann sums defining the Riemann integral of a given function on a given interval must have the same limit, uniqueness of the Riemann integral is built into its definition.
- The conditioning of an integration problem is a measure of the sensitivity to perturbations in the input data. Because integration is an averaging or smoothing process that tends to dampen the effect of small changes in the integrand, integration problems are typically well-behaved with a small condition number.

8.3 Numerical Quadrature

8.3.1 Introduction

In your calculus course you learned to evaluate a definite integral

$$I(f) = \int_b^a f(x)\,dx$$

analytically by finding an **antiderivative** F of the integrand function f, where

$$I(f) = F(b) - F(a).$$

Unfortunately, for some integrals such a closed form does not exist (e.g. $f(x) = \exp(-x^2)$), or they are too complicated to evaluate. In these cases, you can use numerical methods to approximate the value of these integrals.

The numerical approximation of definite integrals is called **numerical quadrature**. This name derives from the ancient methods for approximating areas of irregular shapes by tiling them by small squares, as mentioned above.

To achieve this, we will build upon the definition of Riemann sums: the integral will be approximated by a weighted sum of integrand values at a finite number of sample points in the interval of integration. Specifically, the integral $I(f)$ is approximated by an n-point **quadrature rule**, which has the form

$$Q_n(f) = \sum_{i=1}^{n} w_i f(x_i),$$

where $a \leq x_1 < x_2 < \ldots < x_n \leq b$.

The points x_i at which the integrand f is evaluated are called **nodes** or **abscissas**, and the multipliers w_i are called **weights** or **coefficients**. A quadrature rule is said to be **open** if $a < x_1$ and $x_n < b$, and **closed** if $a = x_1$ and $b = x_n$.

In the next subsections, we will have a look at several methods to choose the nodes and weights with the goal to obtain the best accuracy at the lowest computational cost.

Quadrature rules can be derived using polynomial interpolation. Effectively,

- The integrand function f is evaluated at the points x_i, $i = 1, \ldots, n$.
- The polynomial of degree $n - 1$ that interpolates the function values at those points is determined.
- The integral of the interpolant is then taken as an approximation to the integral of the original function.

To find the weights corresponding to a quadrature rule that integrates the first n polynomial basis functions exactly, we can use the **method of undetermined coefficients**.

If we use the monomial basis, e.g. this strategy results in the following **system of moment equations** with n equations in n unknowns.

General system of moment equations

$$\begin{aligned}
w_1 \cdot 1 + w_2 \cdot 1 + \cdots + w_n \cdot 1 &= \int_a^b 1 \, dx = b - a \\
w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_n \cdot x_n &= \int_a^b x \, dx = (b^2 - a^2)/2 \\
&\vdots \\
w_1 \cdot x_1^{n-1} + w_2 \cdot x_2^{n-1} + \cdots + w_n \cdot x_n^{n-1} &= \int_a^b x^{n-1} \, dx = (b^n - a^n)/n
\end{aligned}$$

In matrix form this becomes:

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_n \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{n-1} & x_2^{n-1} & \cdots & x_n^{n-1} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} b - a \\ (b^2 - a^2)/2 \\ \vdots \\ (b^n - a^n)/n \end{bmatrix}$$

Example

As an example we use the method of undetermined coefficients to derive a three-point quadrature rule

$$Q_3(f) = w_1 f(x_1) + w_2 f(x_2) + w_3 f(x_3)$$

for the interval $[a, b]$ using the monomial basis. We take the two endpoints and the midpoint as the nodes:

- $x_1 = a$
- $x_2 = (a + b)/2$

- $x_3 = b$

This results in the following linear system:

$$\begin{bmatrix} 1 & 1 & 1 \\ a & (a+b)/2 & b \\ a^2 & ((a+b)/2)^2 & b^2 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} b-a \\ (b^2-a^2)/2 \\ (b^3-a^3)/3 \end{bmatrix}$$

Solving this system, we find the weights:

- $w_1 = (b-a)/6$
- $w_2 = 2(b-a)/3$
- $w_3 = (b-a)/6$

The resulting quadrature rule is known as **Simpson's rule** and is an example of a Newton-Cotes quadrature rule.

Accuracy and useful concepts

By construction, an n-point interpolatory quadrature rule integrates each of the first n monomial basis functions exactly, and hence by linearity it integrates any polynomial of degree at most $n-1$ exactly. A quadrature rule is said to be of **degree** d if it is exact (i.e., the error is zero) for every polynomial of degree d but is not exact for some polynomial of degree $d+1$. An n-point interpolatory quadrature rule is of degree at least $n-1$.

The significance of the degree is that it conveniently characterizes the **accuracy** of a given rule. If Q_n is an interpolatory quadrature rule, and p_{n-1} is the polynomial of degree at most $n-1$ interpolating a sufficiently smooth integrand f at the nodes $x_1, \cdots, x_n$, then we get the following rough error bound for the approximate integral:

$$\|I(f) - Q_n(f)\| \leq \frac{1}{4} h^{n+1} \|f^{(n)}\|_\infty$$

where $h = \max\{x_{i+1} - x_i \,:\, i = 1, \ldots, n-1\}$.

the preceding general bound already indicates that we can obtain higher accuracy by taking n larger, or h smaller, or both.

- When the number of sample points is increased, say from n to m, an important factor affecting efficiency is whether the n function values already computed can be reused in the new rule, so that only $m-n$ new function values need be computed. A sequence of quadrature rules is said to be **progressive** if the nodes of Q_{n1} are a subset of those of Q_{n2} for $n_2 > n_1$.
- Instead of (or in addition to) increasing the number of points (and hence the degree), the preceding bound also suggests that the error can be reduced by subdividing the interval of integration into smaller subintervals and applying the quadrature rule separately in each, since this will reduce h. This approach, which is equivalent to using piecewise polynomial interpolation on the original interval, leads to **composite (or compound) quadrature rules**, which we will consider below. For now, we will focus on **simple quadrature rules**, in which a single rule is applied over the entire given interval.

8.3.2 Newton-Cotes Quadrature

Definition and common cases

The simplest placement of nodes for an interpolatory quadrature rule is to choose equally spaced points in the interval $[a, b]$, which is the defining property of **Newton-Cotes quadrature**. An n-point **open Newton-Cotes rule** excludes the edges and has nodes:

$$x_i = a + i(b-a)/(n+1) \quad , \quad i = 1, \cdots, n.$$

An n-point **closed Newton-Cotes rule** includes the edges and has nodes:

$$x_i = a + (i-1)(b-a)/(n-1) \quad , \quad i = 1, \cdots, n.$$

The simplest and best known examples are:

- **The midpoint rule**. Interpolating the function value at the midpoint of the interval by a polynomial of degree zero (i.e., a constant) gives the one-point open Newton-Cotes rule known as **the midpoint rule**:

$$M(f) = (b-a)f\left(\frac{a+b}{2}\right)$$

- **The trapezoid rule** Interpolating the function values at the two endpoints of the interval by a polynomial of degree one (i.e., a straight line) gives the two-point closed Newton-Cotes rule known as **the trapezoid rule**:

$$T(f) = \frac{b-a}{2}(f(a)+f(b))$$

- **Simpson's rule** Interpolating the function values at the two endpoints and the midpoint by a polynomial of degree two (i.e., a quadratic) gives the three-point closed Newton-Cotes rule known as **Simpson's rule**:

$$S(f) = \frac{b-a}{6}\left(f(a)+4f\left(\frac{a+b}{2}\right)+f(b)\right)$$

Example

To illustrate the application of Newton-Cotes quadrature rules, we approximate the integral

$$I(f) = \int_0^1 e^{-x^2}\,dx$$

using each of the three Newton-Cotes quadrature rules just given.

$$\begin{aligned} M(f) &= (1-0)\exp(-0.25) \approx 0.778801 \\ T(f) &= \frac{1}{2}(\exp(0)+\exp(-1)) \approx 0.683940 \\ S(f) &= \frac{1}{6}(\exp(0)+4\exp(-0.25)+\exp(-1)) \approx 0.747180 \end{aligned}$$

The correctly rounded result for this problem is 0.746824.

It is somewhat surprising to see that:

- the magnitude of the error from the trapezoid rule (0.062884) is about twice that from the midpoint rule (0.031977),
- and that Simpson's rule, with an error of only 0.000356, seems remarkably accurate considering the size of the interval over which it is applied.

Error Analysis

The errors of the three Newton-Cotes rules can be analyzed with a Taylor series expansion of the integrand f about the midpoint $m = (a+b)/2$ of the interval $[a, b]$:

$$f(x) = f(m) + f'(m)(x-m) + \frac{f''(m)}{2}(x-m)^2 + \frac{f'''(m)}{6}(x-m)^3 + \frac{f^{(4)}(m)}{24}(x-m)^4 + \cdots$$

Integrating this Taylor series from a to b leads to an exact expression for the integral, when all terms are included. Note that the odd-order terms drop out, yielding:

$$\begin{aligned} I(f) &= f(m)(b-a) + \frac{f''(m)}{24}(b-a)^3 + \frac{f^{(4)}(m)}{1920}(b-a)^5 + \cdots \\ &= M(f) + E(f) + F(f) + \cdots \end{aligned}$$

where $M(f)$, $E(f)$ and $F(f)$ represent the first three terms in the exact expression for the integral. Below, we will write the three Newton-Cotes rules in terms of the Taylor series and compare the outcome to the exact answer. This will reveal the errors of those rules, broken down in different contributions of the form $c_p(b-a)^p$, where p is always an odd power and c_p is a linear coefficient.

- **Midpoint rule**

  The midpoint rule simply coincides with the first term of $I(f)$, namely $M(f)$, meaning that the leading-order term of the error is $E(f)$. Hence, the error is proportional to the width of the interval cubed.

- **Trapezoid rule**

  To derive a comparable error expansion for the trapezoid quadrature rule, evaluate the Taylor series twice, once for $x = a$ and once for $x = b$. Substitute these results into the rule $\frac{f(b)+f(a)}{2}(b-a)$. Then, observe once again that the odd-order terms drop out, and rewrite the leading terms as a function of $M(f)$, $E(f)$ and $F(f)$. This will result in:

  $$T(f) = M(f) + 3E(f) + 5F(f) + \cdots$$

  We may now compute the difference with the exact solution:

  $$T(f) - I(f) = 2E(f) + 4F(f) + \cdots$$

  We may also compute other differences:

  $$T(f) - M(f) = 3E(f) + 5F(f) + \cdots$$

  and hence the difference between the two quadrature rules provides an estimate for the dominant term in their error expansions

  $$E(f) \approx \frac{T(f) - M(f)}{3}$$

  provided that the length of the interval is sufficiently small that $(b-a)^5 \ll (b-a)^3$, and the integrand f is such that $f^{(4)}$ is well-behaved.

  Under these assumptions, we may draw several conclusions from the preceding derivations

  1. The midpoint rule is about twice as accurate as the trapezoid rule, despite being based on a polynomial interpolant of degree one less.
  2. The difference between the midpoint rule and the trapezoid rule can be used to estimate the error in either of them.
  3. Halving the length of the interval decreases the error in either rule by a factor of about 1/8.

- **Simpson's rule**

  An appropriately weighted combination of the midpoint and trapezoid rules eliminates the leading term, $E(f)$, from the error expansion:

  $$S(f) = \frac{2}{3}M(f) + \frac{1}{3}T(f) = M(f) + E(f) + \frac{5}{3}T(f) + \ldots$$

  with error

  $$S(f) - I(f) = \frac{2}{3}F(f) + \cdots$$

  which provides an alternative derivation for Simpson's rule as well as an expression for its dominant error term.

In general, for any odd value of n, an n-point Newton-Cotes rule has degree one greater than that of the polynomial interpolant on which it is based due to cancellation of positive and negative errors.

This implies that the midpoint rule integrates linear polynomials exactly, and hence its degree is one rather than zero. Similarly, the error for Simpson's rule depends on the fourth and higher derivatives in the Taylor expansion, which vanish for cubic as well as quadratic polynomials, so that Simpson's rule is of degree three rather than two (which explains the surprisingly high accuracy obtained in the previous example).

Example

We illustrate these error estimates by computing the approximate value for the integral

$$\int_0^1 x^2\,dx$$

Using the midpoint rule, we obtain

$$M(f) = (1-0)\left(\frac{1}{2}\right)^2 = \frac{1}{4}$$

and using the trapezoid rule we obtain

$$T(f) = \frac{1-0}{2}(0^2 + 1^2) = 1/2$$

Thus, we have the error estimate

$$E(f) \approx \frac{T(f) - M(f)}{3} = \frac{1/4}{3} = \frac{1}{12}$$

We conclude that the error in $M(f)$ is about 1/12 and the error in $T(f)$ is about $-1/6$. In addition, we can now compute the approximate value given by Simpson's rule for this integral,

$$S(f) = \frac{2}{3}M(f) + \frac{1}{3}T(f) = \frac{2}{3}\cdot\frac{1}{4} + \frac{1}{3}\cdot\frac{1}{2} = \frac{1}{3}$$

which is exact for this integral (as is to be expected since, by design, Simpson's rule is exact for quadratic polynomials). Thus, the error estimates for $M(f)$ and $T(f)$ are exact for this integrand (though this would not be true in general).

Closing remarks

Newton-Cotes quadrature rules are relatively easy to derive and to apply, but they have some serious drawbacks. The interpolation of a continuous function at equally spaced points by a high-degree polynomial may suffer from unwanted oscillation, and as the number of interpolation points grows, convergence to the underlying function is not guaranteed.

it can be shown that every n-point Newton-Cotes rule with $n \geq 11$ has at least one negative weight and that the sum of all weights tends to infinity as $n \to \infty$.

This means that Newton-Cotes rules become arbitrarily ill-conditioned, and hence unstable, as the number of points grows. The presence of large positive and negative weights also means that the value of the integral is computed as a sum of large quantities of differing sign, and hence substantial cancellation is likely in finite-precision arithmetic.

For these reasons, we cannot expect to attain arbitrarily high accuracy on a given interval by using a Newton-Cotes rule with a large number of points. In practice, therefore, Newton-Cotes rules are usually restricted to a modest number of points, and if higher accuracy is required, then the interval is subdivided and the rule is applied in each subinterval separately (as we'll see below).

Example integrand

Throughout this notebook, integration algorithms are often tested with the same test case:

$$I = \int_0^1 \exp(-x^2)\,dx$$

The function and the full double precision answer are defined in the following cell:

```
def func(x):
    return np.exp(-(x**2))


EXACT = 0.746824132812427
```

DIY `scipy` integration

The `integrate.newton_cotes` function returns the weights and error coefficient for Newton-Cotes integration with *n* equally spaced data points.

It can therefore be used to compute the integral of a function, using as many points as you wish to evaluate it.

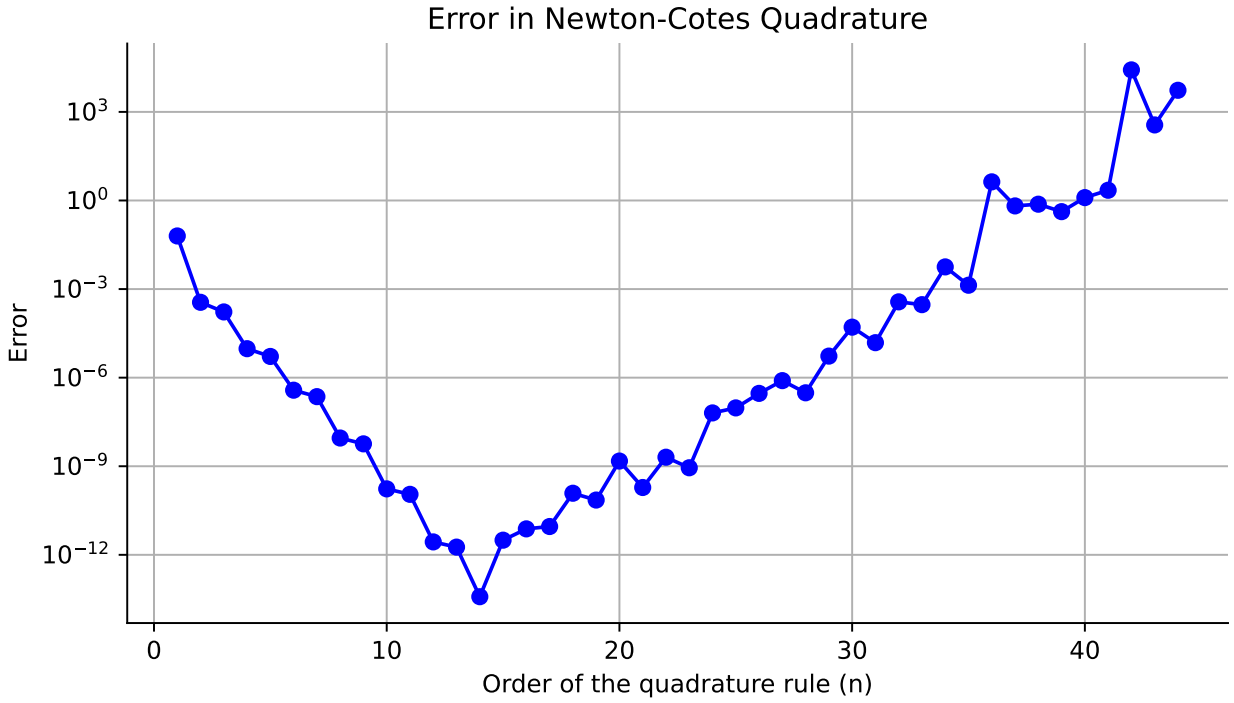
```
def demo_newton_cotes(rn):
    a = 0
    b = 1
    x = np.linspace(a, b, rn + 1)
    an, _B = integrate.newton_cotes(rn, 1)
    dx = (b - a) / rn
    quad = dx * np.sum(an * func(x))
    error = abs(quad - EXACT)
    return quad, error


def demo_newton_cotes_weakness():
    rns = np.arange(1, 45)
    errors = []

    print(f"{'Order':<6} {'Quadrature Result':<18} {'Error'}")
    print("-" * 40)

    for rn in rns:
        quad, error = demo_newton_cotes(rn)
        print(f"{rn:<6} {quad:<18.9f} {error:.5e}")
        errors.append(error)

    # Plotting the error as a function of quadrature order
    plt.close("newton_cotes_weakness")
    fig, ax = plt.subplots(figsize=(7, 4), num="newton_cotes_weakness")
    ax.plot(rns, errors, marker="o", linestyle="-", color="blue")
    ax.set_xlabel("Order of the quadrature rule (n)")
    ax.set_ylabel("Error")
    ax.set_title("Error in Newton-Cotes Quadrature")
    ax.set_yscale("log")
    ax.grid(True)


demo_newton_cotes_weakness()
```

```
Order  Quadrature Result  Error
----------------------------------------
1      0.683939721        6.28844e-02
2      0.747180429        3.56296e-04
3      0.746992320        1.68187e-04
4      0.746833710        9.57704e-06
```

(continues on next page)

(continued from previous page)

```
5       0.746829357         5.22439e-06
6       0.746823757         3.76241e-07
7       0.746823905         2.27545e-07
8       0.746824142         9.02894e-09
9       0.746824139         5.73098e-09
10      0.746824133         1.71191e-10
11      0.746824133         1.11842e-10
12      0.746824133         2.73914e-12
13      0.746824133         1.82465e-12
14      0.746824133         3.81917e-14
15      0.746824133         3.13594e-12
16      0.746824133         7.56706e-12
17      0.746824133         9.09339e-12
18      0.746824133         1.21812e-10
19      0.746824133         7.16401e-11
20      0.746824131         1.49934e-09
21      0.746824133         1.90609e-10
22      0.746824131         2.02248e-09
23      0.746824132         8.87837e-10
24      0.746824197         6.39478e-08
25      0.746824227         9.45534e-08
26      0.746823839         2.93329e-07
27      0.746824926         7.93206e-07
28      0.746823826         3.06958e-07
29      0.746818764         5.36842e-06
30      0.746875425         5.12922e-05
31      0.746839435         1.53020e-05
32      0.746454956         3.69177e-04
33      0.746527431         2.96701e-04
34      0.752460434         5.63630e-03
35      0.745474370         1.34976e-03
36      5.023540499         4.27672e+00
37      1.401483058         6.54659e-01
38      -0.001690437        7.48515e-01
39      0.327903330         4.18921e-01
40      1.998817920         1.25199e+00
41      -1.475183497        2.22201e+00
42      26477.852733333     2.64771e+04
43      -361.289537166      3.62036e+02
44      5397.102441683      5.39636e+03
```



This graph effectively shows that more data points does not always mean better results. The optimal number of data points is in this example about 15, because this produces the smallest error. When we look at the value that corresponds with 15 data points, this is also an accurate approximation of the exact value. 10 to 20 data points also approximate these values well, but as soon as the number of data points exceeds 35, the error becomes very large, very quickly.

8.3.3 Clenshaw-Curtis Quadrature

We saw in the *interpolation* notebook that the Chebyshev points have distinct advantages over equally spaced points for interpolating a continuous function by a polynomial. Similarly, when choosing the nodes in a quadrature rule based on the Chebysev points, we also improve upon the Newton-Cotes rules.

With the Chebyshev points as nodes for a given n, the corresponding weights can again be calculated using the method of undetermined coefficients. It can be shown that the resulting weights are always positive for any n, and that the resulting approximate values converge to the exact integral as $n \to \infty$. Thus, quadrature rules based on the Chebyshev points are extremely attractive in that they are always stable and significantly more accurate than Newton-Cotes rules for the same number of nodes.

Additionally, this type of quadrature rule can be implemented using techniques based on the fast Fourier transform. These efficient implementations of quadrature rules based on the Chebyshev points have become known as **Clenshaw-Curtis quadrature**.

We have seen that Clenshaw-Curtis quadrature rules have many virtues: stability, accuracy, simplicity and progressiveness. Nevertheless, the degree of an n-point rule is only $n - 1$, which is well below the maximum possible. Next, we will see that quadrature rules of maximum degree can be derived by exploiting all of the available degrees of freedom.

8.3.4 Gaussian Quadrature

In all the preceding quadrature rules we chose the n nodes ourselves and then determined the n corresponding weights to maximize the degree of the resulting quadrature rule. With only n parameters free to be chosen, the resulting degree is generally $n - 1$.

If the locations of the nodes were also freely chosen, however, then there would be $2n$ free parameters, so that a degree of $2n - 1$ should be achievable. In Gaussian quadrature, both the nodes and the weights are optimally chosen to maximize the degree of the resulting quadrature rule.

In general, for each n there is a unique n-point Gaussian rule, and it is of degree $2n-1$. Gaussian quadrature rules therefore have the highest possible accuracy for the number of nodes used, but they are significantly more difficult to derive than Newton-Cotes rules.

The nodes and weights can still be determined by the method of undetermined coefficients, but the resulting system of equations is nonlinear.

Example

We will derive a two-point Gaussian quadrature rule on the interval $[-1, 1]$,

$$I(f) = \int_{-1}^{1} f(x)dx \approx w_1 f(x_1) + w_2 f(x_2) = G_2(f)$$

where we can still freely choose w_1, w_2 and x_1 and x_2

The requirement that the first four monomials need to be integrated exactly gives rise to the following equations

$$\begin{aligned} w_1 + w_2 &= \int_{-1}^{1} 1\,dx = 2 \\ w_1 x_1 + w_2 x_2 &= \int_{-1}^{1} x\,dx = 0 \\ w_1 x_1^2 + w_2 x_2^2 &= \int_{-1}^{1} x^2\,dx = 2/3 \\ w_1 x_1^3 + w_2 x_2^3 &= \int_{-1}^{1} x^3\,dx = 0 \end{aligned}$$

Let's solve this nonlinear system using `scipy`

```
def gauss_2point_a_eqs(x):
    """The multidimensional function containing the 4 equations above.

    - `x[0]` and `x[1]` represent $w_1$ and $w_2$.
    - `x[2]` and `x[3]` represent $x_1$ and $x_2$.
    """
    return [
        x[0] + x[1] - 2,
        x[0] * x[2] + x[1] * x[3],
        x[0] * x[2] ** 2.0 + x[1] * x[3] ** 2.0 - 2.0 / 3.0,
        x[0] * x[2] ** 3.0 + x[1] * x[3] ** 3.0,
    ]


# Solve this system with initial guess [1, 1, 1, 1].
optimize.root(gauss_2point_a_eqs, [1, 1, 1, 1]).x
```

```
array([1., 1., 1., 1.])
```

As shown above, one solution for this system is given by

- $x_1 = -1/\sqrt{3} \approx -0.57735027$
- $x_2 = 1/\sqrt{3} \approx 0.57735027$

- $w_1 = 1$
- $w_2 = 1$

Thus the two-point Gaussian quadrature rule has the form

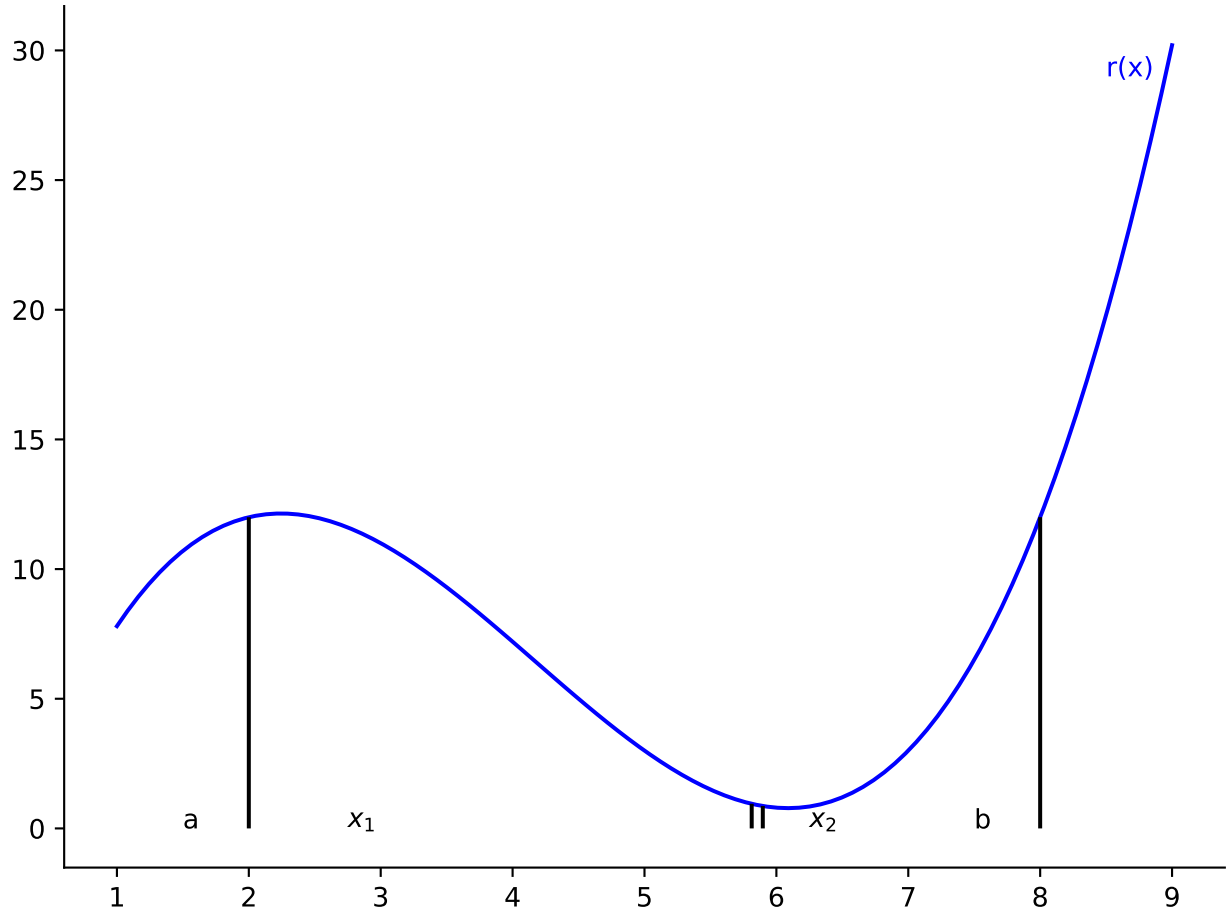
$$G_2(f) = f(-1/\sqrt{3}) + f(1/\sqrt{3})$$

and by construction it has degree three.

Another example on how Gaussion Quadrature works, but now for the interval $[2, 8]$ and plotted for the integrand $f(x) = \frac{1}{15}(6x - 3)(x - 5)(x - 7) + 3$

```
def gauss_2point_b_eqs(x):
    return [
        x[2] + x[3] - 6,
        x[2] * x[0] + x[3] * x[1] - 30,
        x[2] * x[0] ** 2 + x[3] * x[1] ** 2 - 168,
        x[2] * x[0] ** 3 + x[3] * x[1] ** 3 - 1020,
    ]


def example_gc28():
    def integrand(x):
        return 1 / 15 * (6 * x - 3) * (x - 5) * (x - 7) + 3

    solution = optimize.root(gauss_2point_b_eqs, [2, 2, 1, 1]).x
    print(solution)
    ts = np.linspace(1, 9, 100)
    plt.close("gc28")
    fix, ax = plt.subplots(num="gc28")
    ax.plot(ts, integrand(ts), color="Blue")
    ax.vlines(x=2, ymin=0, ymax=integrand(2), color="black")
    ax.vlines(x=8, ymin=0, ymax=integrand(8), color="black")
    ax.vlines(x=solution[0], ymin=0, ymax=integrand(solution[0]), color="black
↪")
    ax.vlines(x=solution[1], ymin=0, ymax=integrand(solution[1]), color="black
↪")
    ax.text(1.5, 0, "a")
    ax.text(7.5, 0, "b")
    ax.text(2.75, 0, "$x_1$")
    ax.text(6.25, 0, "$x_2$")
    ax.text(8.5, 29, "r(x)", color="Blue")


example_gc28()
```

```
[ 5.81289175  5.8970391  -6.20816465 10.91915352]
```



Properties

- These examples are typical in that for any n the Gaussian **nodes are symmetrically** placed about the midpoint of the interval; for odd values of n the midpoint itself is always a node.
- This example is also typical in that the **nodes are usually irrational numbers**, even if the endpoints a and b are rational. This feature makes Gaussian rules relatively inconvenient to calculate by hand, compared to simple Newton-Cotes rules. Usually, tabulated values are used for Gaussian quadrature.
- Gaussian quadrature rules are also harder to apply than Newton-Cotes rules because the weights and nodes are derived for some specific interval and thus any other interval of integration must be transformed into the standard interval for which the nodes and weights have been tabulated.

If we want to use a quadrature rule that is tabulated on the interval $[\alpha, \beta]$,

$$\int_{\alpha}^{\beta} f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

to approximate an integral on the interval $[a, b]$,

$$I(g) = \int_{a}^{b} g(t)\,dt$$

then we must use a change of variable from x in $[\alpha, \beta]$ to t in $[a, b]$. Many such transformations are possible, but a simple linear transformation

$$t = \frac{(b-a)x + a\beta - b\alpha}{\beta - \alpha}$$

has the advantage of preserving the degree of the quadrature rule. The integral is then given by

$$I(g) \approx \frac{b-a}{\beta-\alpha} \sum_{i=1}^{n} w_i g\left(\frac{(b-a)x_i + a\beta - b\alpha}{\beta-\alpha}\right)$$

Example

To illustrate a change of interval, we use the two-point Gaussian quadrature rule G_2 derived for the interval $[-1, 1]$ in the previous example to approximate the integral

$$I(g) = \int_0^1 e^{-t^2}\, dt$$

Using the linear transformation of variable just shown, we have

$$t = \frac{x+1}{2}$$

so that the integral is approximated by $G_2(g)$=

$$\frac{1}{2}\left[\exp\left(-\left(\frac{(-1/\sqrt{3})+1}{2}\right)^2\right) + \exp\left(-\left(\frac{(1/\sqrt{3})+1}{2}\right)^2\right)\right] \approx 0.746595$$

which is slightly more accurate than the result given by Simpson's rule for this integral, despite using only two points instead of three.

This algorithm is implemented in `integrate.fixed_quad` which "Integrate func from a to b using Gaussian quadrature of order n."

```
def demo_quadrature():
    """Illustrate `integrate.quadrature` with the integral int_0^1 exp(-x**2)␣
 ↪dx."""
    y, _ = integrate.fixed_quad(func, 0, 1, n=2)
    print(f"integral = {y:.10f}")


demo_quadrature()
```

```
integral = 0.7465946883
```

In the figure above you see an example of a lineair transformation of the variable. This is needed because if you work with variable t, the integral has boundaries 0 and 1, this is not tabulated for Gaussian quadrature. When you work with variable x, the boundaries are given by -1 and 1, these values are tabulated for Gaussian quadrature and hence are simple to work with.

By design, Gaussian quadrature rules have maximal degree, and hence optimal accuracy, for the number of points used. Moreover, it can be shown that the resulting weights are always positive for any n, so that Gaussian quadrature rules are always stable and the resulting approximate values converge to the exact integral as $n \to \infty$.

Unfortunately, Gaussian quadrature rules also have a serious drawback:

for $m \neq n$, G_m and G_n have no nodes in common (except for the midpoint when m and n are both odd). Thus,

> Gaussian rules are **not progressive**, which means that when the number of nodes is increased, say from n to m, m new evaluations of the integrand are required rather than $m - n$.

Avoiding this additional work is the motivation for **Kronrod quadrature rules**. Such rules come in pairs: an n-point Gaussian rule G_n and a $(2n + 1)$-point Kronrod rule K_{2n+1} whose nodes are optimally chosen subject to the constraint that all of the nodes of G_n are reused in K_{2n+1}. Thus, n of the nodes used in K_{2n+1} are prespecified, leaving the remaining $n + 1$ nodes, as well as all $2n + 1$ of the weights (including those corresponding to the nodes of G_n), free to be chosen to maximize the degree of the resulting rule. The rule K_{2n+1} is therefore of degree $3n+1$, whereas a true $(2n+1)$-point Gaussian rule would be of degree $4n + 1$. Thus, there is a tradeoff between accuracy and efficiency.

One of the main reasons for using two quadrature rules with different numbers of points is to obtain an error estimate for the approximate value of the integral based on the difference between the values given by the two rules. In using a Gauss-Kronrod pair, the value of K_{2n+1} is taken as the approximation to the integral, and a realistic but conservative estimate for the error, based partly on theory and partly on experience, is given by

$$(200\|G_n - K_{2n+1}\|)^{1.5}$$

Because they efficiently provide both high accuracy and a reliable error estimate, Gauss-Kronrod rules are among the most effective quadrature methods available, and they form the basis for many of the quadrature routines in major software libraries. The pair of rules (G_7 , K_{15}), in particular, has become a commonly used standard

8.3.5 Composite Quadrature

Thus far we have considered simple quadrature rules obtained by interpolating the integrand function by a single polynomial over the entire interval of integration.

The accuracy of such a rule can be increased, and the error estimated, by increasing the number of interpolation points, and hence the corresponding degree of the polynomial interpolant.

An alternative is to subdivide the original interval into two or more subintervals and apply a simple quadrature rule in each subinterval. Summing these partial results then yields an approximation to the overall integral.

Such an approach is equivalent to using piecewise polynomial interpolation on the original interval and then integrating the piecewise interpolant to approximate the integral.

A **composite**, or **compound**, quadrature rule on a given interval $[a, b]$ results from subdividing the interval into k subintervals, typically of uniform length $h = (b-a)/k$, applying an n-point simple quadrature rule Q_n in each subinterval, and then taking the sum of these results as the approximate value of the integral.

If the rule Q_n is open, then evaluating the composite rule will require kn evaluations of the integrand function. If Q_n is closed, on the other hand, then some of the points are repeated, so that only $k(n - 1) + 1$ evaluations of the integrand are required.

> **Example**
>
> If the interval $[a, b]$ is subdivided into k subintervals of length $h = (b - a)/k$ and $x_j = a + jh$, $j = 0, \ldots, k$, then the **composite midpoint rule** is given by
>
> $$M_k(f) = \sum_{j=1}^{k} (x_j - x_{j-1}) f\left(\frac{x_{j-1} + x_j}{2}\right) = h \sum_{j=1}^{k} f\left(\frac{x_{j-1} + x_j}{2}\right)$$
>
> and the **composite trapezoid rule** is given by
>
> $$T_k(f) = \sum_{j=1}^{k} \frac{(x_j - x_{j-1})}{2} \left(f(x_{j-1}) + f(x_j)\right) = h\left(\frac{1}{2}f(a) + f(x_1) + \cdots + f(x_{k-1}) + \frac{1}{2}f(b)\right)$$

- In principle, by taking k sufficiently large it is possible to achieve arbitrarily high accuracy (up to the limit of the arithmetic precision) using a composite rule, even with an underlying rule Q_n of low degree, although this may not be the most efficient way to attain a given level of accuracy.
- Composite quadrature rules also offer a particularly simple means of estimating the error by using different levels of subdivision, which can easily be made progressive.

Composite midpoint rule demonstration

```
def plot_adaptive_general(num):
    fig, ax = plt.subplots(num=num, clear=True)
    a, b = 0, 1

    # Generate x values for plotting the function
    x_vals = np.linspace(a, b, 1000)
    y_vals = func(x_vals)

    # Plot the function for the midpoint rule
    ax.plot(x_vals, y_vals, label=r"$e^{-x^2}$", color="green")

    ax.set_xlabel("x")
    ax.set_ylabel("f(x)")
    ax.axhline(0, color="black", linewidth=0.5, linestyle="--")

    return ax, a, b


def plot_midpoint_rule(n_subintervals_midpoint):
    ax, a, b = plot_adaptive_general("midpoint_rule")

    # Calculate midpoints and function values for the midpoint rule.
    midpoints_midpoint = np.linspace(
        a + (b - a) / (2 * n_subintervals_midpoint),
        b - (b - a) / (2 * n_subintervals_midpoint),
        n_subintervals_midpoint,
    )
    midpoint_values = func(midpoints_midpoint)

    # Highlight the midpoints and corresponding function values
    # for the midpoint rule.
    ax.scatter(
        midpoints_midpoint,
        midpoint_values,
        color="red",
        label=f"Midpoints (Intervals: {n_subintervals_midpoint})",
    )

    # Draw rectangles representing the midpoint rule
    # for the selected number of subintervals.
    for j in range(n_subintervals_midpoint):
        rect = plt.Rectangle(
            (a + j * (b - a) / n_subintervals_midpoint, 0),
            (b - a) / n_subintervals_midpoint,
            midpoint_values[j],
            alpha=0.3,
            color="blue",
```

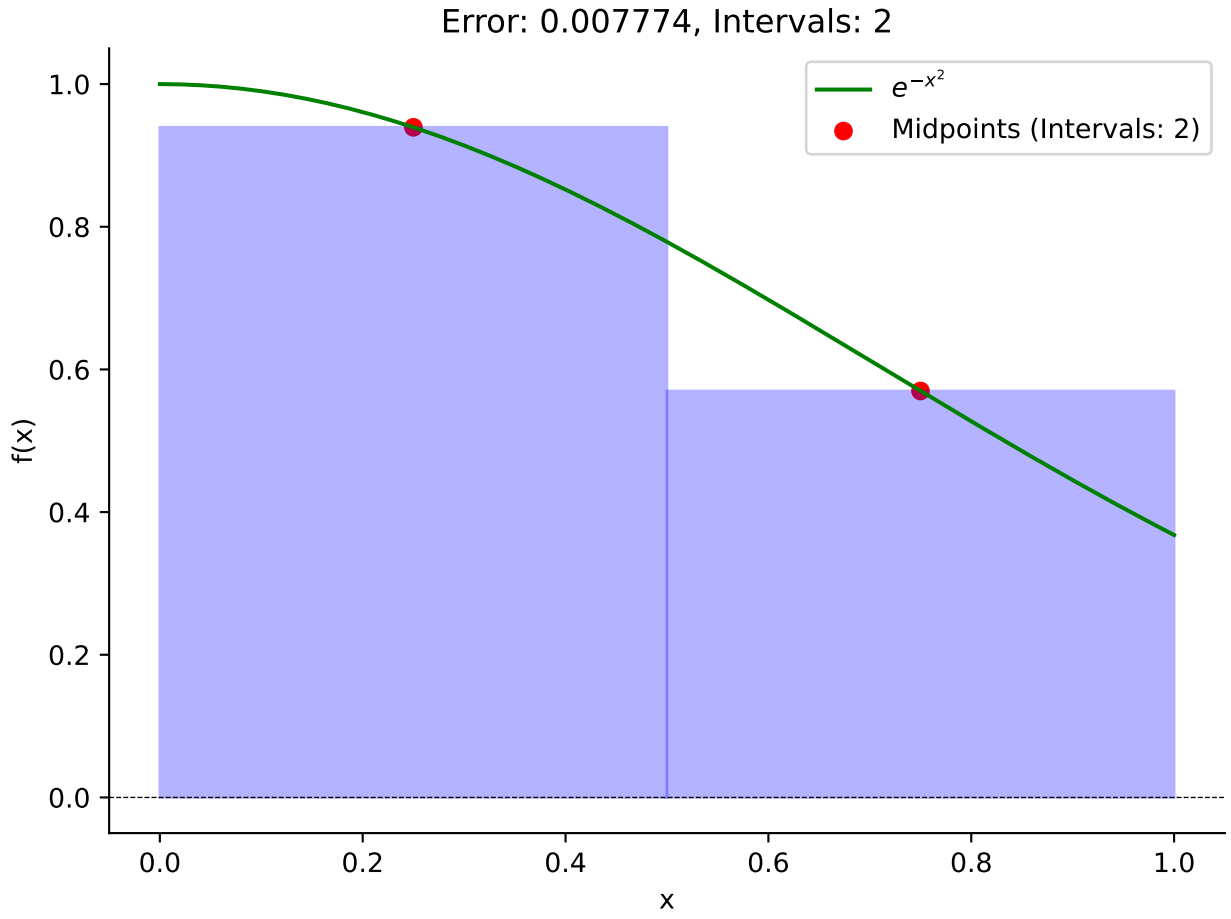
(continues on next page)

(continued from previous page)

```
        )
        ax.add_patch(rect)

    # Calculate the exact integral
    # and the numerical approximation using quad function.
    numerical_integral = np.sum(midpoint_values) * (b - a) / n_subintervals_
↪midpoint

    # Calculate the error between the exact integral and the numerical␣
↪approximation.
    error = numerical_integral - EXACT
    # Update the title with the error information.
    ax.set_title(f"Error: {error:.6f}, Intervals: {n_subintervals_midpoint}")
    ax.legend()


# Use the interact function to update the plot based on the slider value.
plt.close("midpoint_rule")
interact(
    plot_midpoint_rule,
    n_subintervals_midpoint=widgets.IntSlider(
        value=2, min=1, max=20, step=1, description="Intervals"
    ),
)
```

```
interactive(children=(IntSlider(value=2, description='Intervals', max=20,␣
↪min=1), Output()), _dom_classes=('wi…
```

```
<function __main__.plot_midpoint_rule(n_subintervals_midpoint)>
```



Composite trapezoid illustration

```
def plot_trapezoid_rule(n_subintervals_trapezoid):
    ax, a, b = plot_adaptive_general("trapezoid_rule")

    # Calculate points for the trapezoidal rule.
    x_points = np.linspace(a, b, n_subintervals_trapezoid + 1)
    y_points = func(x_points)

    # Highlight the endpoints and corresponding function values
    # for the trapezoidal rule.
    ax.scatter(
        x_points,
        y_points,
        color="red",
        label=f"Endpoints (Intervals: {n_subintervals_trapezoid + 1})",
    )

    # Draw trapezoids representing the trapezoidal rule
    # for the selected number of subintervals.
    for j in range(n_subintervals_trapezoid):
        x_left = x_points[j]
        x_right = x_points[j + 1]
        y_left = y_points[j]
```

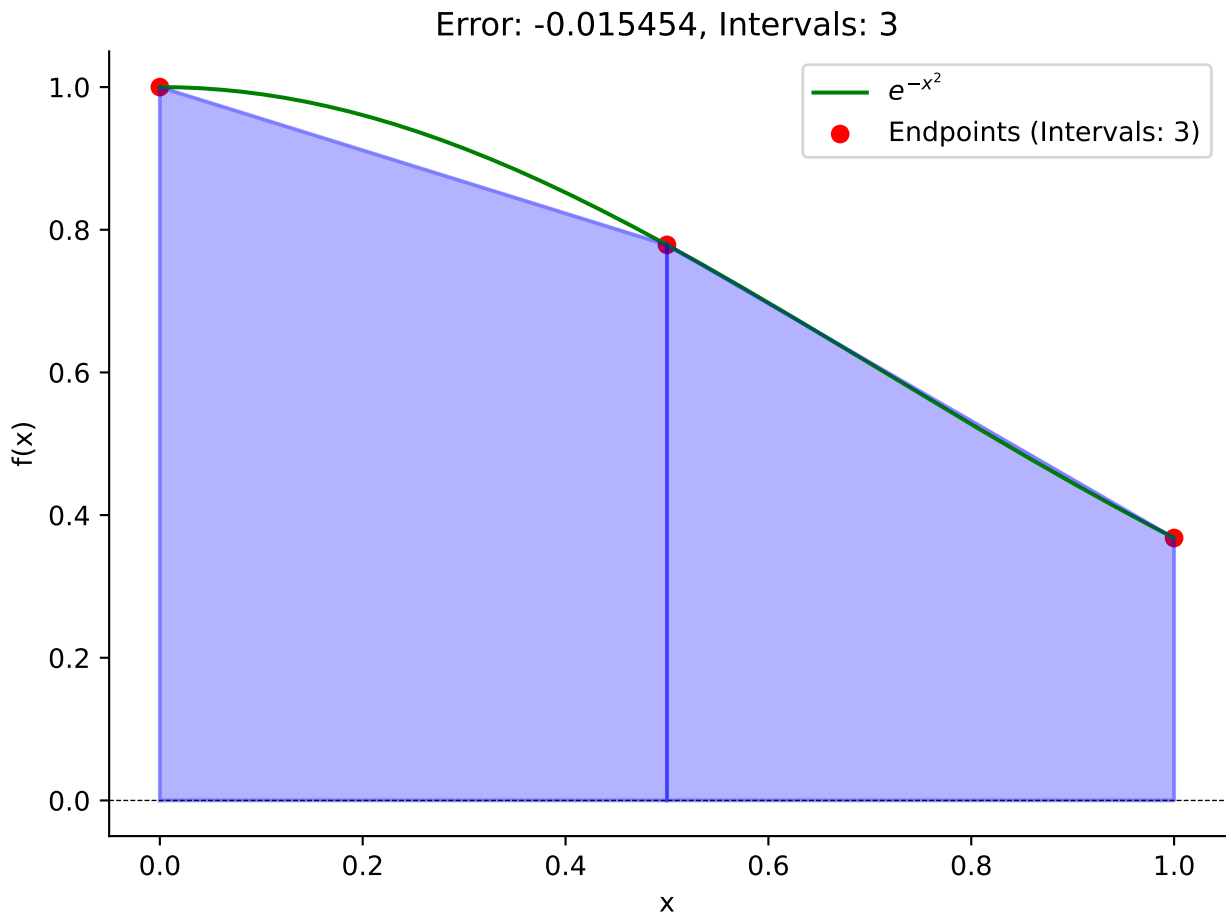
(continues on next page)

(continued from previous page)

```
        y_right = y_points[j + 1]

        # Draw trapezoid
        ax.plot(
            [x_left, x_right, x_right, x_left, x_left],
            [0, 0, y_right, y_left, 0],
            color="blue",
            alpha=0.3,
        )
        ax.fill_between(
            [x_left, x_right], [y_left, y_right], color="blue", alpha=0.3
        )

    # Calculate the exact integral and the numerical approximation
    # using quad function.
    numerical_integral = np.trapezoid(y_points, x_points)

    # Calculate the error between the exact integral and the numerical␣
↪approximation
    error = numerical_integral - EXACT

    # Update the title with the error information
    ax.set_title(f"Error: {error:.6f}, Intervals: {n_subintervals_trapezoid +␣
↪1}")
    ax.legend()


# Use the interact function to update the plot based on the slider value
plt.close("trapezoid_rule")
interact(
    plot_trapezoid_rule,
    n_subintervals_trapezoid=widgets.IntSlider(
        value=2, min=1, max=20, step=1, description="Intervals"
    ),
)
```

```
interactive(children=(IntSlider(value=2, description='Intervals', max=20,␣
↪min=1), Output()), _dom_classes=('wi…
```

```
<function __main__.plot_trapezoid_rule(n_subintervals_trapezoid)>
```



A composite quadrature rule with an error estimate suggests a simple automatic quadrature procedure: continue subdividing all the subintervals until the estimated overall error meets the desired accuracy tolerance.

However, maintaining uniform subdivisions is grossly inefficient for many integrands, however, as large numbers of function evaluations may be expended in regions where the integrand function is well behaved and the accuracy tolerance is easily met. A more intelligent approach is **adaptive quadrature**, in which the interval of integration is selectively refined to reflect the behavior of any particular integrand function.

A typical adaptive quadrature strategy works as follows:

1. First we need a pair of quadrature rules, say Q_{n1} and Q_{n2}, whose difference provides an error estimate. A few examples are

- The trapezoid and midpoint rules, whose difference overestimates the error in the more accurate rule by a factor of three (see above).
- Greater efficiency is usually obtained with rules of higher degree, however, such as the Gauss-Kronrod pair (G_7 ,K_{15}).
- Another alternative is to use a single rule at two different levels of subdivision; Simpson's rule is a popular choice in this approach.

In any case, to minimize the number of function evaluations required, the pair of rules should be progressive.

The adaptive procedure is now conceptually simple:

2. apply both rules Q_{n1} and Q_{n2} on the initial interval of integration $[a, b]$.
3. If the resulting approximate values for the integral differ by more than the desired tolerance, divide the interval into two or more subintervals and repeat the procedure on each subinterval.

4. If the tolerance is met on a given subinterval, then no further subdivision of that subinterval will be required.
5. If the tolerance is not met on a given subinterval, then the subdivision process is repeated again, and so on until the tolerance is met on all subintervals.

Such a strategy leads to a nonuniform sampling of the integrand function that places many sample points in regions where the function is difficult to integrate and relatively few points where the function is easily integrated.

This conceptually simple explanation hides several practical implementation issues, [e.g. How should the stopping criterion be implemented? For example, should the error tolerance be relative (usually preferable), or absolute (in case the value of the integral is near zero), or a combination of the two?], which we will not concern ourselves with here.

The following code gives an example of an adaptive quadrature rule using the midpoint and trapezoid rules. This structrure is based on section 8.3.6, p.355 of Micheal Heath's book 'Scientific computing - an introductory survey (revised second edition)'.

```
def adaptquad(func, a, b, tol=10**-6):
    """
    Calculate definite integrals based on adaptive quadrature.

    Parameters
    ----------
    func : callable
        The function to be integrated from a to b.
    a, b : float
        The integration limits.
    tol : float, optional
        Desired error tolerance.

    Returns
    ----------
    I2 : float
        Calculated value of the integral
    """
    I1 = (b - a) * func(a + (b - a) / 2)  # midpoint rule
    I2 = (b - a) * (func(a) + func(b)) / 2  # trapezoid rule
    m = a + (b - a) / 2
    if m <= a or m >= b:  # stopping criterion
        print("Warning: Tolerance may not be met.")
        return I2
    estimated_err = abs(I1 - I2) / 3
    if estimated_err < tol:
        return I2
    else:
        # call adaptquad recursively for each subinterval
        return adaptquad(func, a, m, tol / 2) + adaptquad(func, m, b, tol / 2)


def demo_adaptquad():
    a = 0
    b = np.pi
    result = adaptquad(np.sin, a, b)
    error = np.abs(2 - result)  # comparison with the analytical solution
    print(f"Adaptive quadrature = {result}")
    print(f"Error on adaptive quadrature = {error}")


demo_adaptquad()
```

```
Adaptive quadrature = 1.9999989760564243
Error on adaptive quadrature = 1.0239435757064541e-06
```

8.4 Other integration problems

8.4.1 Tabular data

Thus far we have assumed that the integrand function can be evaluated at any desired point within the interval of integration. This assumption may not be valid if the integrand is defined only by a table of its values at selected discrete points, as is typical of empirical measurements, for example.

A reasonable approach to integrating such tabular data is by piecewise interpolation. For example, integrating the piecewise linear interpolant to tabular data gives a composite trapezoid rule.

An excellent method for integrating tabular data is provided by Hermite cubic or cubic spline interpolation. In effect, the overall integral is computed by integrating analytically each of the cubic pieces that make up the interpolant.

As an example of how to perform such integrations with `scipy`, we'll integrate tabular data containing 10 equally spaced function evaluations between 0 and 1 of the function

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

```
def demo_tabular(n):
    """Demo of integrate.simpson to integrate tabular data.

    Note that, as the number of samples, n, increases,
    the integration tends towards the theoretical result of
    int_0^1 exp(-x^2) = 0.7468241328901555.

    Also note that the accuracy is much better for even numbers
    of points (odd number of intervals) because Simpsons rule
    requires this, and falls back on the less accurate trapezoidal
    rule in case of an even number of data points.
    """
    x = np.linspace(0.0, 1.0, num=n)
    y = func(x)
    return integrate.simpson(y, x)


for i in np.arange(2, 12):
    result = demo_tabular(i)
    err = np.abs(result - EXACT)
    print(f"{i:2d}  {result:18.15f}  {err:9.2e}")
```

```
 2   0.683939720585721   6.29e-02
 3   0.747180428909510   3.56e-04
 4   0.748781989423117   1.96e-03
 5   0.746855379790987   3.12e-05
 6   0.747030542060102   2.06e-04
 7   0.746830391489345   6.26e-06
 8   0.746871199329973   4.71e-05
 9   0.746826120527467   1.99e-06
10   0.746839922191154   1.58e-05
11   0.746824948254444   8.15e-07
```

8.4.2 Improper integrals

Boundedness of both the integrand function and the interval of integration are inherent in the definition of the Riemann integral. If either the integrand (containing a singularity) or the interval (at least one of the limits is ∞) is unbounded, then it may still be possible to define an improper integral.

Integrating an integral with unbounded interval of integration:

- Replace any infinite limit of integration by a finite value. Such a finite limit should be chosen carefully so that the omitted tail is negligible or its contribution to the integral can be estimated. But the remaining finite interval should not be so wide that an adaptive quadrature routine will be fooled into sampling the integrand badly.
- Transform the variable of integration so that the new interval is finite. Typical transformations include $x = -\log t$ or $x = t/(1-t)$. Care must be taken not to introduce singularities or other difficulties by such a transformation.
- Use a quadrature rule, such as Gauss–Laguerre or Gauss–Hermite, that is designed for an unbounded interval.

Integrating an integral with singularities

For an integrand having an integrable singularity within the interval of integration, one may be tempted simply to try an adaptive quadrature routine and hope that it will work, but such an approach is unlikely to prove satisfactory. Outright failure will result if the integrand happens to be evaluated at the singularity, which will likely occur if the singularity lies at one of the endpoints, as singularities often do. Even if the routine is lucky enough to avoid evaluating the integrand at the singularity, an adaptive quadrature routine will generally be extremely inefficient for an integrand having a singularity because polynomials, which never have vertical asymptotes, cannot efficiently approximate functions that do (recall that our error bounds depend on higher derivatives of the integrand, which will inevitably be large near a singularity).

A better approach for dealing with a singularity in the integrand is to remove the singularity either by transforming the variable of integration or by dividing out or subtracting off an analytically integrable function having the same singularity.

There is often some art involved in finding such transformations, and not all singularities are removable in this manner. If there is more than one singularity, then the transformations required to remove them may conflict, in which case a remedy is to break the interval of integration into subintervals, each of which contains at most one singularity, typically at one endpoint.

8.4.3 Double integrals

Thus far we have considered only one-dimensional integrals, where we wish to determine the area under a curve over an interval. In evaluating a two-dimensional, or double integral, we wish to compute the volume under a surface over a planar region.

For a rectangular region $[a, b] \times [c, d]$, a double integral has the form

$$\int_a^b \int_c^d f(x, y)\, dx\, dy$$

The practical way of approximating such integrals is shown using the `scipy` example below

> **Example**
>
> Consider the exponential integral (with an unbounded integration interval):
>
> $$E_n(x) = \int_1^\infty \frac{e^{-xt}}{t^n}\, dt$$
>
> This can be calculated using `scipy.quad`:

```
def expt_integrand(t, x, n=3):
    return np.exp(-x * t) / t**n


def expt_int1(x, n=3):
    return integrate.quad(expt_integrand, 1, np.inf, args=(x, n))[0]
```

> The function which is integrated itself can also use the quad argument (though the error bound may underestimate the error due to possible numerical error in the integrand from the use of quad).
>
> The integral in this case is
>
> $$I_n = \int_0^\infty \int_1^\infty \frac{e^{-xt}}{t^n}\, dt\, dx = \frac{1}{n}$$

```
integrate.quad(expt_int1, 0, np.inf)
```

```
(0.33333333325010883, 2.8604069921197956e-09)
```

The mechanics for double and triple integration have been wrapped up into the functions `dblquad` and `tplquad`. These functions take the function to integrate and four, or six arguments, respectively. The limits of all inner integrals can be defined as functions.

```
integrate.dblquad(expt_integrand, 0, np.inf, 1, np.inf)
```

```
(0.33333333325010883, 1.3888461883425516e-08)
```

> Here is an example for a triple integral:
>
> $$I = \int_0^1 \int_0^1 \int_0^1 \exp(x)\, yz\, dx\, dy\, dz$$
>
> The analytical solution is
>
> $$\frac{e-1}{4} = 0.4295704571$$

```
integrate.tplquad(lambda x, y, z: np.exp(x) * y * z, 0, 1, 0, 1, 0, 1)
```

```
(0.4295704571147614, 1.899400317474637e-14)
```

8.4.4 Multiple integrals

To evaluate a multiple integral in dimensions higher than two, the method above for double integrals still work in principle (and is used in `nquad`), but their cost grows rapidly with the number of dimensions.

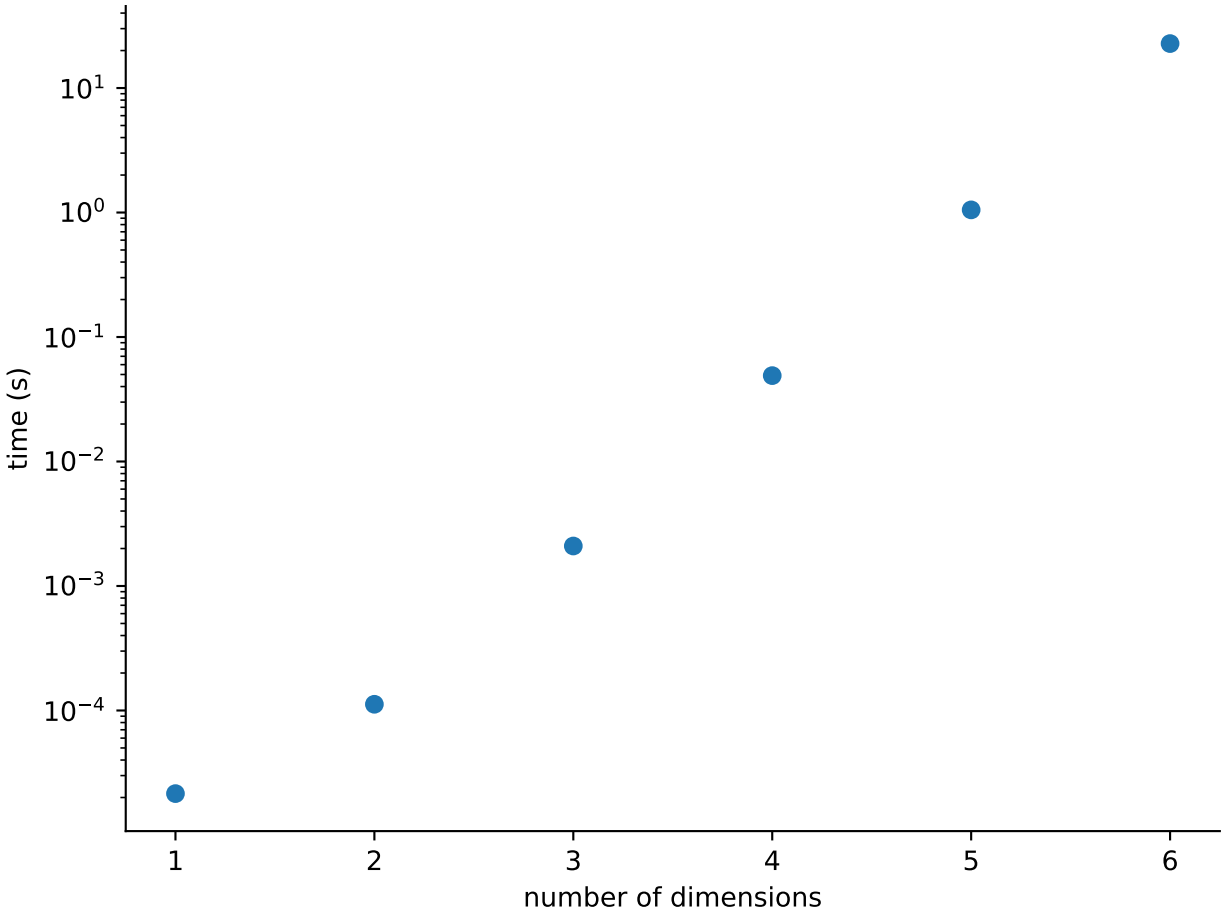
The only generally viable approach for computing integrals in higher dimensions is the **Monte Carlo method**. The function is sampled at n points distributed randomly in the domain of integration, and then the mean of these function values is multiplied by the area (or volume, etc.) of the domain to obtain an estimate for the integral. The error in this estimate goes to zero as $1/\sqrt{n}$.

The Monte Carlo method is not competitive for integrals in one or two dimensions, but the beauty of the method is that its convergence rate is independent of the number of dimensions. Thus, for example, one million points in six

dimensions amounts to only ten points per dimension, which is much better than any type of conventional quadrature rule would require for the same level of accuracy.

> Let us use the following integral as an example:
>
> $$\int_0^1 \int_0^1 \ldots \int_0^1 x_1 + x_2 + \ldots + x_N \, dx_1 \, dx_2 \, \ldots \, dx_N$$
>
> In one dimension this becomes:
>
> $$\int_0^1 x_1 \, dx_1$$
>
> In two dimensions:
>
> $$\int_0^1 \int_0^1 x_1 + x_2 \, dx_1 \, dx_2$$
>
> And so on. We'll look at the computational time needed to integrate this integral for the first 5 dimensions.

```
def measure_time(n):
    name = str(f"integrate.nquad((lambda *x_args: sum(x_args)), [(0, 1)]*{n})
 ↪")
    return timeit.timeit(stmt=name, setup="from scipy import integrate",␣
 ↪number=1)


def demo_multidim(n):
    x = np.arange(1, n + 1)
    y = np.zeros(n)

    for i in np.arange(1, n + 1):
        y[i - 1] = measure_time(i)

    plt.close("multidim")
    fig, ax = plt.subplots(num="multidim")
    ax.set_yscale("log")
    ax.plot(x, y, "o")
    ax.set_xlabel("number of dimensions")
    ax.set_ylabel("time (s)")
    ax.set_xticks(np.arange(1, n + 1))


demo_multidim(6)
```



The plot shows that the computational cost grows exponentially and becomes very high when the dimension is >=5.

8.5 Numerical Differentiation

Let's touch upon differentiation now. In contrast to integration, differentiation is an inherently sensitive problem, as small perturbations in the data can cause large changes in the result.

Generally, when approximating the derivative of a function whose values are known only at a discrete set of points, a good approach is to fit some smooth function to the given discrete data and then differentiate the approximating function to approximate the derivatives of the original function.

8.5.1 Finite Difference Approximations

Although finite difference formulas are generally inappropriate for discrete or noisy data, they are very useful for approximating derivatives of a smooth function that is known analytically, or can be evaluated accurately for any given argument, or is defined implicitly by a differential equation.

The following equations will be useful for the next notebooks, where we'll deal with the numerical solution of differential equations.

We want to approximate the first and second derivatives of a smooth function $f : \mathbb{R} \to \mathbb{R}$ at a point x.

For a given step size h, we consider the Taylor series expansions

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

and

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

Approximations for the first derivative

- Solving the first series for $f'(x)$, we obtain the **forward difference formula**

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

  This approximation is first-order accurate since the dominant remainder of the series is $\mathcal{O}(h)$.

- Similarly, we obtain the **backward difference formula** from the second series:

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

- Adding both series together gives the **centered difference formula**

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

  which is second-order accurate.

Approximations for the second derivative

- Finally, subtracting the second series from the first gives a centered difference formula for the second derivative

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} - \frac{f^{(4)}(x)}{12}h^2 + \cdots \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

  which is also second-order accurate.

By using function values at additional points, $x \pm 2h$, $x \pm 3h$,..., we can derive similar finite difference approximations with still higher accuracy or for higher-order derivatives, although these come at a cost of more function evaluations.

8.6 Richardson Extrapolation

In many problems, such as numerical integration or differentiation, we compute an approximate value for some quantity based on some step size. Ideally, we would like to obtain the limiting value as the step size goes to zero, but we cannot take the step size to be arbitrarily small because of excessive cost or rounding error. Based on values for nonzero step sizes, however, we may be able to estimate what the value would be for a step size of zero. Note that the extrapolated value, is still only an approximation to the exact solution, and its accuracy is still limited by the step size and arithmetic precision used.

Let $F(h)$ denote the value obtained with step size h, using a method for which we know the scaling behavior as $h \to 0$ (i.e. the order of the method).

Starting from

$$F(h) = a_0 + a_1 h^p + \mathcal{O}(h^r)$$

as $h \to 0$ for some p and r, with $r > p$.

We assume that we know the values of p and r from the method used to obtain $F(h)$, but not a_0 or a_1 (because $a_0 = F(0)$ is the quantity we are trying to find in the first place!).

Suppose that we have computed F for two step sizes, say, h and h/q for some positive integer q. Then we have

$$F(h) = a_0 + a_1 h^p + \mathcal{O}(h^r)$$

and

$$F(h/q) = a_0 + a_1(h/q)^p + \mathcal{O}(h^r) = a_0 + a_1 q^{-p} h^p + \mathcal{O}(h^r)$$

This system of two linear equations in the two unknowns a_0 and a_1 is easily solved to obtain

$$a_0 = F(h) + \frac{F(h) - F(h/q)}{q^{-p} - 1} + \mathcal{O}(h^r)$$

Thus, the accuracy of the improved value, a_0 , is $\mathcal{O}(h^r)$ rather than $\mathcal{O}(h^p)$.

If $F(h)$ is known for several values of h, then the extrapolation process can be repeated to produce still more accurate approximations, up to the limitations imposed by finite-precision arithmetic.

Example

To illustrate Richardson extrapolation, we use it to improve the accuracy of a finite difference approximation to the derivative of the function $\sin(x)$ at the point $x = 1$. Using the first-order accurate forward difference formula (FD), we have for this problem

$$F(h) = a_0 + a_1 h + \mathcal{O}(h^2),$$

which means that $p = 1$ and $r = 2$ in this case. Using step sizes of $h = 0.5$ and $h/2 = 0.25$ (corresponding to $q = 2$), we find

$$\text{FD with stepsize h: } \frac{d\sin(x)}{dx} \approx \frac{\sin(x+h) - \sin(x)}{h} = \frac{\sin(1.5) - \sin(1)}{0.5} = 0.312048 = F(h)$$

and

$$\text{FD with stepsize h/2: } \frac{d\sin(x)}{dx} \approx \frac{\sin(x+h/2) - \sin(x)}{h/2} = \frac{\sin(1.25) - \sin(1)}{0.25} = 0.430055 = F(h/2)$$

The extrapolated value is then given by

$$F(0) = a_0 = F(h) + \frac{F(h) - F(h/2)}{(1/2) - 1} = 2F(h/2) - F(h) = 0.548061$$

For comparison, the real result is given by $\cos(1) \approx 0.540302$

8.6.1 Romberg integration

Another example of Richardson extrapolation is **Romberg Integration**.

Similar to the estimate of the derivative of the function at a certain value in the previous example, our strategy to estimate the value of an integral is as follows

- Use a scheme (e.g. the composite trapezoid quadrature rule) to estimate our integral given a certain step size h
- Repeat this for a second step size (for example $h/2$)
- The two obtained values can be considered as two points $F(h)$ and $F(h/2)$ of the function F whose behavior is determined by the scaling properties of the used integration scheme
- Calculate $F(0)$ to estimate what the value of our integral would be for an infinitely small h

Example

Consider the integral

$$\int_0^{\pi/2} \sin(x)\, dx$$

If we use the composite trapezoid quadrature rule, we get the following equation: (remember from the error analysis section that in the trapezoid rule the $\mathcal{O}(h^2)$ cancel out).

$$F(h) = a_0 + a_1 h^2 + \mathcal{O}(h^4)$$

with $h = \pi/2$, we obtain the value $F(h) = 0.785398$.

This was obtained in the following way:

The composite trapezoid rule is given by:

$$\begin{aligned} T_k(f) &= \sum_{j=1}^{k} \frac{x_j - x_{j-1}}{2}(f(x_{j-1}) + f(x_j)) \\ &= h(\frac{1}{2}f(a) + f(x_1) + \cdots + f(x_{k-1} + \frac{1}{2}f(b)) \end{aligned}$$

For our specific function $\sin(x)$ with $h = \pi/2$ this becomes:

$$F(\pi/2) = \frac{\pi}{2}\left(\frac{1}{2}\sin(0) + \frac{1}{2}\sin(\pi/2)\right) = 0.785398$$

Taking $q = 2$, we obtain the value $F(h/2) = F(\pi/4) = 0.948059$.

This was obtained in the following way:

$$F(\pi/4) = \frac{\pi}{4}\left(\frac{1}{2}\sin(0) + \sin(\pi/4) + \frac{1}{2}\sin(\pi/2)\right) = 0.948059$$

The extrapolated value is then given by

$$F(0) = a_0 + \frac{F(h) - F(h/2)}{2^{-2} - 1} = \frac{4F(h/2) - F(h)}{3} = 1.002280$$

which is substantially more accurate than either value previously computed (the exact answer is 1).

For any integer $k \leq 0$, let $T_{k,0}$ denote the approximation to the integral $\int_a^b f(x)dx$ given by the composite trapezoid rule with step size $h_k = (b-a)/2^k$. Then for any integer j, $j = 1, \ldots, k$, we can recursively define the successive extrapolated values

$$T_{k,j} = \frac{4^j T_{k,j-1} - T_{k-1,j-1}}{4^j - 1}$$

which form a triangular array

$$\begin{array}{llllll} T_{0,0} & & & & \\ T_{1,0} & T_{1,1} & & & \\ T_{2,0} & T_{2,1} & T_{2,2} & & \\ T_{3,0} & T_{3,1} & T_{3,2} & T_{3,3} & \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$$

In this example we have already computed $T_{0,0} = 0.785398$, $T_{1,0} = 0.948059$, and the extrapolated value $T_{1,1} = 1.002280$. If we reduce the step size by another factor of two in the composite trapezoid rule, we obtain $T_{2,0} = F(h/4) = F(\pi/8) = 0.987116$.

We can now combine the results for $h/2$ and $h/4$ to obtain the extrapolated value

$$T_{2,1} = F(h/2) + \frac{F(h/2) - F(h/4)}{2^{-2} - 1} = \frac{4T_{2,0} - T_{1,0}}{4 - 1} = 1.000135$$

Because we have eliminated the leading $\mathcal{O}(h^2)$ error term for the composite trapezoid rule, the accuracy of the first level of extrapolated values is $\mathcal{O}(h^4)$. Thus, we can further extrapolate on these values, but now with $p = 4$, to obtain

$$T_{2,2} = \frac{4^2 T_{2,1} - T_{1,1}}{4^2 - 1} = \frac{16 \times 1.000135 - 1.002280}{15} = 0.999992$$

which is still more accurate than any of the values computed previously.

Recursive computation of extrapolated values in this manner, based on the composite trapezoid rule with successively halved step sizes, is called **Romberg integration**. It is capable of producing very high accuracy (up to the limit imposed by the arithmetic precision) for very smooth integrands. It is also implemented in `scipy.integrate.romberg`.

The same example can be solved with SciPy's `romb` function:

```
x = np.linspace(0, np.pi / 2, 33)
integrate.romb(np.sin(x), x[1] - x[0], show=True)
```

```
Richardson Extrapolation Table for Romberg Integration
======================================================
 0.78540
 0.94806  1.00228
 0.98712  1.00013  0.99999
 0.99679  1.00001  1.00000  1.00000
 0.99920  1.00000  1.00000  1.00000  1.00000
 0.99980  1.00000  1.00000  1.00000  1.00000  1.00000
======================================================
```

```
np.float64(1.0000000000000002)
```

8.7 SciPy

Further information on all the functions implemented in scipy related to integration can be found here: scipy.integrate

CHAPTER 9

Ordinary Differential Equations (ODEs)

```
import fractions

import matplotlib.pyplot as plt
import numpy as np
from scipy import integrate, linalg, optimize
```

9.1 Introduction and useful concepts

Most physical systems change over time. From an orbiting satellite to a cooling cup of coffee, from a swinging pendulum to a decaying radioisotope, from reacting chemical species to competing biological species, a state of flux is the norm.

One of the motivating problems for the invention of differential calculus was to characterize the motion of celestial bodies and earthly projectiles so that their future locations could be predicted. Differential equations provide a mathematical language for describing continuous change.

Beginning with Newton's laws of motion, most of the fundamental laws of science are expressed as differential equations.

Suppose that the state of a system at any given time t is described by some vector function $\mathbf{y}(t)$,

For example, the components of $\mathbf{y}(t)$ might represent the spatial coordinates of a projectile or concentrations of various chemical species.

A **differential equation** prescribes a relationship between this unknown state function $\mathbf{y}(t)$ and one or more of its derivatives with respect to t that must hold at any given time.

In solving a differential equation, the objective is to determine a differentiable function $\mathbf{y}(t)$ that satisfies the prescribed relationship.

Finding such a solution of the differential equation is important because it will enable us to predict the future evolution of the system over time.

> **Example**
>
> *Newton's Second Law of Motion* states that force equals mass times acceleration ($F = ma$)

> This differential equation relates the state of an object, in this case its position in space, to the second derivative of that state function. In one dimension, the differential equation looks like this:
>
> $$F(t, y(t), dy(t)/dt) = m\frac{d^2y}{dt^2}$$
>
> where the force F in general depends on the time t, the position $y(t)$, and the velocity $dy(t)/dt$, and the acceleration is the second derivative of the position $\frac{d^2y}{dt^2}$. If F is the gravitational force on an object (near Earth's surface), then $F = -mg$, where g is the standard gravity. The solution to the differential equation is then given by
>
> $$y(t) = -\frac{1}{2}gt^2 + c_1t + c_2$$
>
> where c_1 and c_2 are constants that depend on the initial position and velocity of the object. This solution function describes the trajectory of the object over time under the force of gravity.

When there is only one independent variable, such as time, then all derivatives of the dependent variables are with respect to that independent variable, and we have an **ordinary differential equation, or ODE**.

In the PDE notebook we will consider systems with more than one independent variable, so that partial derivatives are required and we have a **partial differential equation, or PDE**.

Notation: To make ODEs less cumbersome to express, we will use the notation $\mathbf{y}'(t) = d\mathbf{y}(t)/dt$ to indicate the first derivative with respect to the (only) independent variable t, and we will often suppress the explicit dependence on t, for example writing $\mathbf{y}' = d\mathbf{y}/dt$, with the dependence on t understood.

> **Example**
>
> With these conventions Newton's Second Law can be written $F = my''$.

The highest-order derivative appearing in an ODE determines the **order** of the ODE.

> **Example**
>
> Newton's Second Law is a second-order ODE.

The most general kth order ODE has the **implicit** form

$$\mathbf{f}(t, \mathbf{y}, \mathbf{y}', \ldots, \mathbf{y}^{(k)}) = \mathbf{0}$$

where $\mathbf{f}$ is a known function and y(t) is to be determined.

A kth order ODE is said to be **explicit** if it can be written in the form

$$\mathbf{y}^{(k)}(t) = \mathbf{f}(t, \mathbf{y}, \mathbf{y}', \ldots, \mathbf{y}^{(k-1)})$$

Many ODEs arise naturally in this form, and many others can be transformed into it.

> **Example**
>
> Newton's Second Law is technically implicit, but it can be made explicit by dividing both sides by the mass m, so that it becomes
>
> $$y'' = F/m$$

We will only consider first-order ODEs. This is not a real restriction because a higher-order ODE can always be transformed into an equivalent first-order system as follows.

For an explicit kth order ODE of the form just given, define the k new unknowns

- $u_1(t) = y(t)$,
- $u_2(t) = y'(t)$
- $\cdots$
- $u_k(t) = y^{(k-1)}(t)$,

so that the original kth order equation becomes a system of k first-order equations:

$$\begin{bmatrix} \mathbf{u}_1' \\ \mathbf{u}_2' \\ \vdots \\ \mathbf{u}_{k-1}' \\ \mathbf{u}_k' \end{bmatrix} = \begin{bmatrix} \mathbf{u}_2 \\ \mathbf{u}_3 \\ \vdots \\ \mathbf{u}_k \\ \mathbf{f}(t, \mathbf{u}_1, \mathbf{u}_2, \cdots, \mathbf{u}_k) \end{bmatrix} = \mathbf{g}(t, \mathbf{u})$$

Example

Again, Newton's Second Law, which is of second order, is a good example.

If we define the new unknowns:

- $u_1(t) = y(t)$
- $u_2(t) = y'(t)$

,then Newton's Second Law becomes a system of two first-order equations

$$\begin{bmatrix} u_1' \\ u_2' \end{bmatrix} = \begin{bmatrix} u_2 \\ F/m \end{bmatrix}$$

Which looks a lot more familiar if we introduce the velocity v instead of u_2 and the position x instead of u_1

$$\begin{bmatrix} x' \\ v' \end{bmatrix} = \begin{bmatrix} v \\ F/m \end{bmatrix}$$

An ODE $\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$ does not by itself determine a unique solution function because only the slopes $\mathbf{y}'(t)$ of the solution components are prescribed by the ODE for any value of t, not the solution value $\mathbf{y}(t)$ itself, so there is usually an infinite family of functions that satisfy the ODE.

To single out a particular solution, we must specify the value of the solution function, denoted by $\mathbf{y}_0$, for some value of t, denoted by t_0. Thus, part of the given problem data is the requirement that $\mathbf{y}(t_0) = \mathbf{y}_0$.

Under reasonable assumptions, this additional requirement determines a unique solution to the given ODE. Because the independent variable t often represents time, we think of t_0 as the initial time and $\mathbf{y}_0$ as the initial value of the state vector.

Accordingly, the requirement that $\mathbf{y}(t_0) = \mathbf{y}_0$ is called an **initial condition**, and an ODE together with an initial condition is called an **initial value problem**, or **IVP**.

Starting from its initial state $\mathbf{y}_0$ at time t_0 , the ODE governs the dynamic evolution of the system for $t \geq t_0$, and we seek a function $\mathbf{y}(t)$ that satisfies the initial condition and describes the state of the system as a function of time.

If we integrate the ODE $\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$ and use the initial condition $\mathbf{y}(t_0) = \mathbf{y}_0$, we obtain the integral equation

$$\mathbf{y}(t) = \mathbf{y}_0 + \int_{t_0}^{t} \mathbf{f}(s) ds$$

which explains, why solving an ODE, by whatever means, is often referred to as **integrating the ODE**.

9.2 Numerically solving ODE's

Our approach to solving differential equations numerically will be based on discretization:

We will replace differential equations by algebraic equations whose solutions approximate those of the given differential equations. For an initial value problem, approximate solution values are generated step by step in discrete increments across the interval in which the solution is sought. For this reason, numerical methods for solving ODEs are sometimes called discrete variable methods

A numerical solution of an IVP is obtained by starting at time t_0 with the given initial value $\mathbf{y}_0$ and attempting to track the solution trajectory dictated by the ODE.

We can determine the initial slope $\mathbf{y}'_0$ of each component of the solution by evaluating $\mathbf{f}$ at the given initial data, i.e., $\mathbf{y}'_0 = \mathbf{f}(t_0, \mathbf{y}_0)$. We use this information to predict the value $\mathbf{y}_1$ of the solution at some future time $t_1 = t_0 + h_0$ for some suitably chosen increment h_0. We can then evaluate $\mathbf{y}'_1 = \mathbf{f}(t_1, \mathbf{y}_1)$ and repeat the process to take another step forward, and so on until we reach the final desired time.

9.2.1 Euler forward method

The simplest example of this approach is **Euler's method**, for which the approximate solution at time $t_{k+1} = t_k + h_k$ is given by

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(t_k, \mathbf{y}_k)$$

Euler's method is an example of a **single-step method** in which the next approximate solution value depends only on the current values of t_k , $\mathbf{y}_k$, and h_k.

For reasons we will soon see, Euler's method is generally inefficient, so it is seldom used in practice, but it is of fundamental importance in understanding the basic concepts and principles in solving differential equations numerically

Euler's method can be derived in several ways:

- **Finite diffference appoximation**

If we replace the derivative $\mathbf{y}'(t)$ in the ODE $\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$ by a first-order forward difference approximation (see notebook on integration and differentiation), we obtain an algebraic equation

$$\frac{\mathbf{y}_{k+1} - \mathbf{y}_k}{h_k} = \mathbf{f}(t_k, \mathbf{y}_k)$$

which gives Euler's method when solved for y_{k+1}.

- **Taylor Series**

Consider the Taylor series

$$\mathbf{y}(t + h) = \mathbf{y}(t) + h\mathbf{y}'(t) + 1/2h^2\mathbf{y}''(t) + \cdots$$

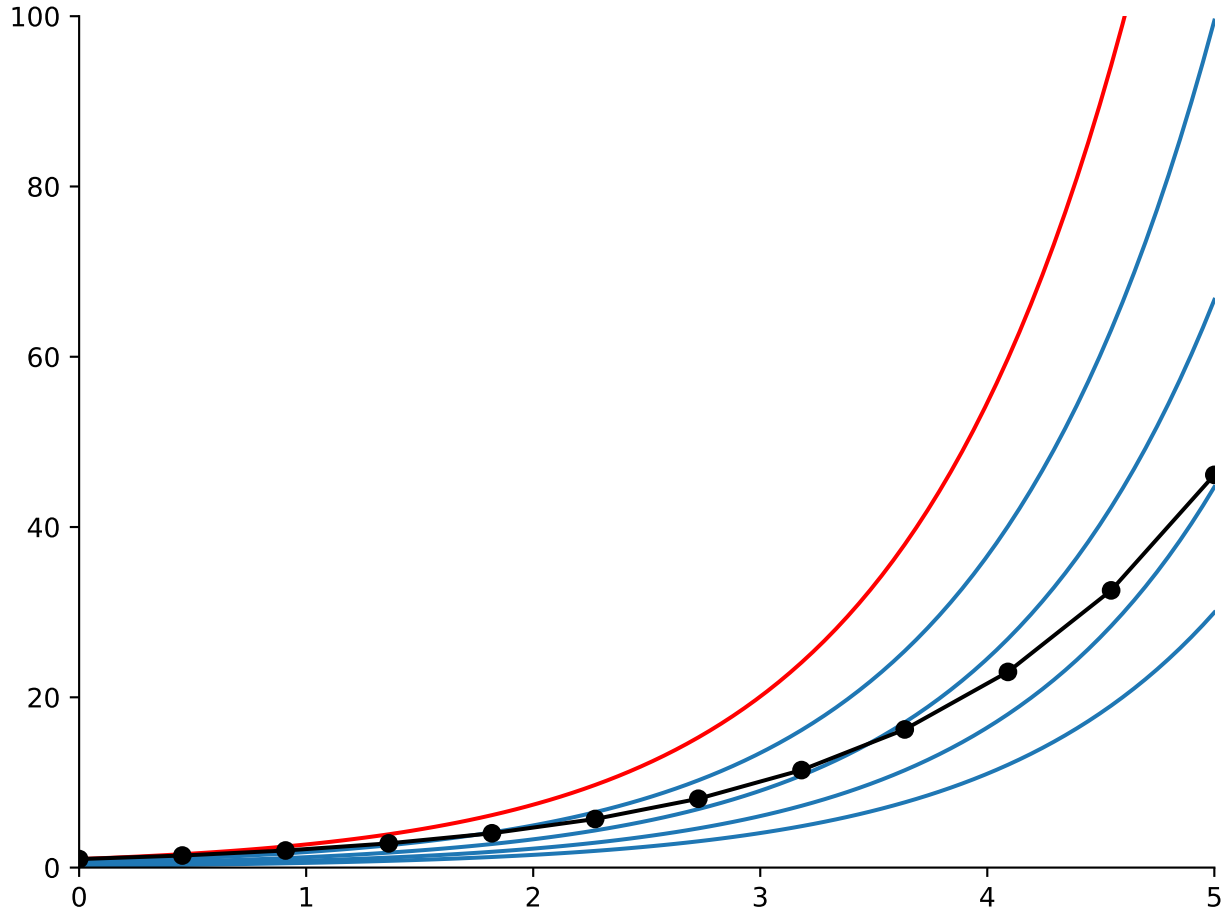
Euler's method results from taking $t = t_k$, $h = h_k$, $\mathbf{y}'(t_k) = \mathbf{f}(t_k, \mathbf{y}_k)$, and dropping terms of second and higher order.

This is actually a very convenient and powerful method to construct solvers and we'll see a more advanced example further below when we construct our very own Runge-Kutta solver.

> **Example**
>
> Consider the ODE $y' = y$ with initial value y_0 at $t_0 = 0$
>
> This simple problem is easily solved analytically, but for illustration let us apply Euler's method to solve it numerically. For simplicity, we will use a fixed step size h. We first advance the solution from time $t_0 = 0$ to time $t_1 = t_0 + h$
>
> $$y_1 = y_0 + hy'_0 = y_0 + hy_0 = (1 + h)y_0$$

> Note that the approximate solution value y_1 we obtain at t_1 is not exact (i.e., $y_1 \neq y(t_1)$).
>
> For example, if $t_0 = 0$, $y_0 = 1$, and $h = 0.5$, then $y_1 = 1.5$, whereas the exact solution for this initial value is $y(0.5) = \exp(0.5) \approx 1.649$.
>
> Thus, the value y_1 lies on a different solution of the ODE from the one on which we started, as illustrated below.

```
def demo_unstable_solution():
    """Illustrate the (in)stability of the solution of the ODE y'=y with␣
 ↪y(0)=1
    when integrated with a sufficiently small step size h=0.5.
    """

    def exact(x, c):
        return np.exp(x - c)

    x = np.linspace(0, 5, 100)
    steps: int = 12
    t = np.linspace(0, 5, steps)
    h = (t[0] + t[-1]) / steps
    y = np.zeros(steps)
    y[0] = 1
    for i in np.arange(1, steps):
        y[i] = y[i - 1] + h * y[i - 1]

    plt.close("unstable")
    fig, ax = plt.subplots(num="unstable")
    for i in np.linspace(0, 1.6, 5):
        ax.plot(
            x, exact(x, i), "-", c="#1f77b4" if i != 0 else "r", label="y(0)␣
 ↪= 1"
        )

    ax.plot(t, y, "o-", c="k")
    ax.axis([0, 5, 0, 100])


demo_unstable_solution()
```



We can continue to take additional steps, generating a table of discrete values of the approximate solution over whatever interval we desire. As we do so, we will hop from one solution to another at each step. The solutions of this ODE are unstable, so the errors we make at each step are amplified with time as a result of the divergence of the solutions, as could be seen in the previous figure

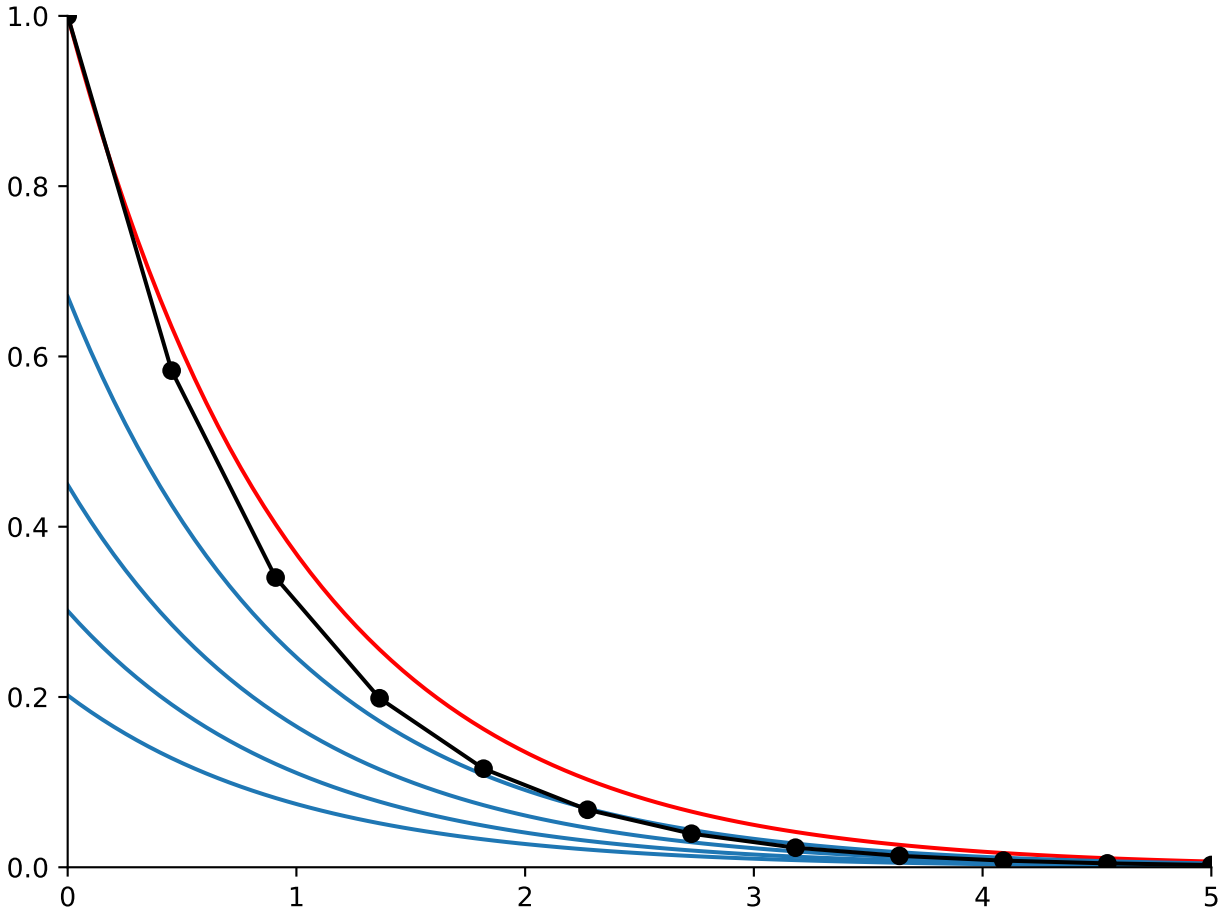
For an equation with stable solutions, on the other hand, the errors in the numerical solution do not grow, and for an equation with asymptotically stable solutions, such as $y' = -y$, the errors diminish with time, as shown in the next figure.

```
def demo_stable_solution():
    """Illustrate the stability of the solution of the ODE y'=-y with y(0)=1
    when integrated with a sufficiently small step size h=0.5
    """

    def exact(x, c):
        return np.exp(-x - c)

    x = np.linspace(0, 5, 100)
    steps: int = 12
    t = np.linspace(0, 5, steps)
    h = (t[0] + t[-1]) / steps
    y = np.zeros(steps)
    y[0] = 1
    for i in np.arange(1, steps):
        y[i] = y[i - 1] - h * y[i - 1]
```

(continues on next page)

(continued from previous page)

```
    plt.close("stable")
    fig, ax = plt.subplots(num="stable")
    for i in np.linspace(0, 1.6, 5):
        ax.plot(
            x, exact(x, i), "-", c="#1f77b4" if i != 0 else "r", label="y(0)␣
↪= 1"
        )

    ax.plot(t, y, "o-", c="k")
    ax.axis([0, 5, 0, 1])


demo_stable_solution()
```



9.2.2 Accuracy and Stability

Rounding error and truncation error

Like other methods that approximate derivatives by finite differences, a numerical procedure for solving an ODE suffers from two distinct sources of error:

- **Rounding error** , which is due to the finite precision of floating-point arithmetic
- **Truncation error** (or **discretization error**), which is due to the method used, and which would remain, even if all arithmetic were performed exactly

Although they arise from different sources, these two types of errors are not independent of each other. For example, the truncation error can usually be reduced by using a smaller step size h, but doing so may incur greater rounding error.

In most practical situations, truncation error is the dominant factor in determining the accuracy of numerical solutions of ODEs, so we will ignore rounding error in this context.

Local and global error

The truncation error at the kth step comes in two distinct but related flavors:

- **Global error** is the cumulative overall error

$$\mathbf{e}_k = \mathbf{y}_k - \mathbf{y}(t_k)$$

where $\mathbf{y}_k$ is the computed solution at t_k and $\mathbf{y}(t)$ is the true solution of the ODE passing through the initial point $(t_0, \mathbf{y}_0)$.

- **Local error** is the error made in one step of the numerical method,

$$\mathscr{e}_k = \mathbf{y}_k - \mathbf{u}_{k-1}(t_k)$$

where $\mathbf{u}_{k-1}$ is the solution of the ODE passing through the previous point $(t_{k-1}, \mathbf{y}_{k-1})$.

The global error is obviously of primary interest, but only the local error can be readily estimated and controlled, so we need to understand the relationship between the two.

In a bank savings account earning compound interest, early deposits have more time to grow than later ones, and this growth means that the total value of the account is not simply the sum of the individual deposits. Similarly, the global error of an approximate solution to an ODE at a given step reflects not only the local error at that step, but also the compounded effects of the local errors at all previous steps. Thus, the global error is not simply the sum of the local errors. If the solutions of the ODE are diverging, then the local errors at each step are magnified over time, so that the global error is greater than the sum of the local errors. If the solutions of the ODE are converging, on the other hand, then the global error may be less than the sum of the local errors. In order to assess the effectiveness of a numerical method, we need to characterize both its local error (accuracy) and the compounding effects over multiple steps (stability).

Accuracy

The accuracy of a numerical method is said to be of order p if

$$\mathscr{e}_k = \mathcal{O}(h_k^{p+1})$$

The motivation for this definition, with the order of accuracy one less than the exponent of the step size in the local error, is that if the local error is $\mathcal{O}(h_k^{p+1})$, then the local error per unit step, $\mathscr{e}_k/h_k$, is $\mathcal{O}(h_k^p)$, and it can be shown that under reasonable conditions the global error $\mathbf{e}_k$ is $\mathcal{O}(h^p)$, where h is the average step size.

Stability

The concept of stability of a numerical method for an ODE is analogous to the stability of solutions to an ODE:

Recall that a solution to an ODE is stable if perturbations of the solution do not diverge away from it over time. Similarly, a numerical method is said to be stable if small perturbations do not cause the resulting numerical solution to diverge away without bound.

Such divergence of numerical solutions could be caused by instability of the solution to the ODE, but as we will see, it can also be caused by the numerical method itself, even when the solutions to the ODE are stable.

To focus specifically on instability due to the numerical method, an alternate definition of stability requires that the numerical solution at any arbitrary but fixed time t remains bounded as $h \to 0$. The two definitions are effectively equivalent, however, as either definition prohibits excessive growth as the number of steps becomes arbitrarily large

> **example**
>
> Let us first examine stability and accuracy in the simple context of Euler's method applied to the scalar ODE
>
> $$y' = \lambda y$$

where λ is a (possibly complex) constant.

With initial condition $y(0) = y_0$, the exact solution to the IVP is given by

$$y(t) = y_0 e^{\lambda t}$$

Applying Euler's method to this ODE using a fixed step size h, we have the recurrence

$$y_{k+1} = y_k + h\lambda y_k = (1 + h\lambda)y_k$$

which implies that

$$y_k = (1 + h\lambda)^k y_0$$

The quantity $1 + h\lambda$ is called the **growth factor**.

- If $\mathrm{Re}(\lambda) < 0$, then the exact solution of the ODE decays to zero as t increases, as will the successive computed solution values if $\|1 + h\lambda\| < 1$.
- If $\|1 + h\lambda\| > 1$, on the other hand, then the computed solution values grow without bound regardless of the sign of $\mathrm{Re}(\lambda)$, which means that Euler's method can be unstable even when the exact solution is stable.

In order for Euler's method to be stable, the step size h must satisfy the inequality

$$\|1 + h\lambda\| \leq 1$$

which says that $h\lambda$ must lie inside a circle in the complex plane of radius 1 centered at -1.

If λ is real, then $h\lambda$ must lie in the interval $(-2, 0)$, which means that for $\lambda < 0$, we must have $h \leq -2/\lambda$ for Euler's method to be stable.

We also note that the growth factor 1 + hλ agrees with the series expansion

$$e^{h\lambda} = 1 + h\lambda + \frac{(h\lambda)^2}{2} + \frac{(h\lambda)^3}{6} + \cdots$$

through terms of first order in h, so the accuracy of Euler's method is of first order.

This stability criterion is illustrated in the demo below:

```
def demo_euler_stability():
    """This demo illustrates the stability of the Euler forward method when
    integrating y'=-y with  y(0)=10 from 0 to 10000 as function of the step
    size taken in this interval.

    We expect the method to be stable if 0<h<2,
    and print the error for h=1.99,2.00 and 2.01.
    """

    def int(h):
        y = 10.0
        tmax = 10000.0
        for _t in np.arange(0.0, tmax + h, h):
            y = y - h * y
        return np.abs(y - 10 * np.exp(-tmax))

    print("h\t error")
    for h in np.arange(1.99, 2.011, 0.01):
        print(h, "\t", int(h))
```

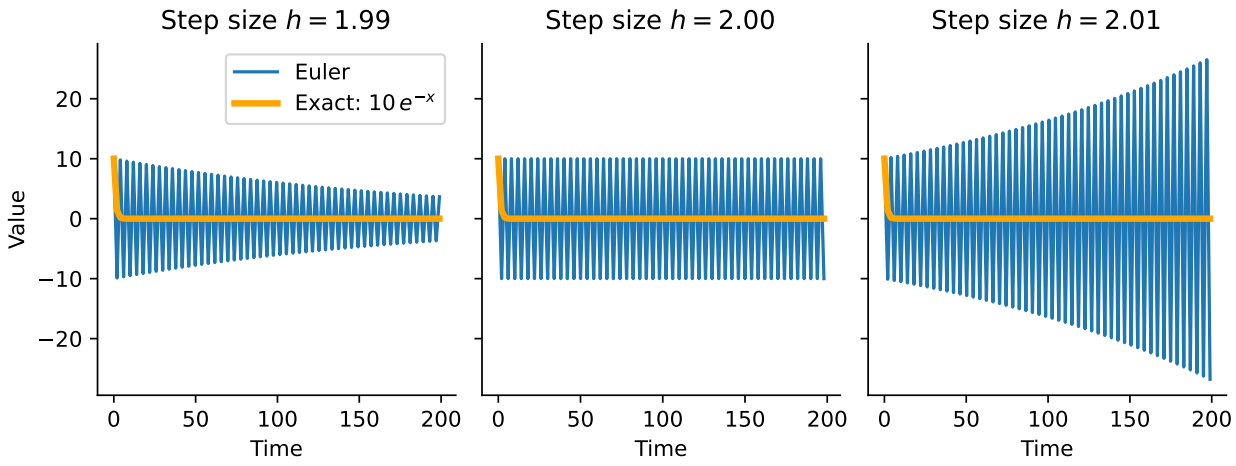
(continues on next page)

(continued from previous page)

```
demo_euler_stability()
```

```
h         error
1.99          1.1432027010231947e-21
2.0          10.0
2.01          3.217194489886198e+22
```

```
def plot_euler_stability():
    """Plot the results of the Euler method for different step sizes
    and compare with the exact solution.

    This function creates a 1x3 grid of subplots to display the numerical
    approximation and exact solution for each step size(h) using the Euler
    forward method.
    """
    plt.close("euler_stab")
    fig, axs = plt.subplots(1, 3, figsize=(8, 3), sharey=True, num="euler_stab
 ↪")

    h_values = np.arange(1.99, 2.011, 0.01)

    for idx, h in enumerate(h_values):
        y = 10
        y_list = [y]
        t_values = np.arange(0, 200, h)
        for _t in t_values:
            y -= h * y
            y_list.append(y)

        axs[idx].plot(t_values, y_list[:-1], label="Euler")
        axs[idx].plot(
            t_values,
            10 * np.exp(-t_values),
            label=r"Exact: $10\, e^{-x}$",
            linestyle="-",
            color="orange",
            linewidth=3,
        )
        axs[idx].set_title(f"Step size $h = {h:.2f}$")
        axs[idx].set_xlabel("Time")
        if idx == 0:
            axs[idx].legend()
            axs[idx].set_ylabel("Value")


plot_euler_stability()
```



General case

A more general analysis produces the same stability and accuracy results as we obtained using the simple scalar test equation. Especially for more complicated numerical methods, this simple scalar test ODE is far easier to work with than a general ODE, and it produces essentially the same stability results if we equate the complex coefficient λ with the eigenvalues of the Jacobian matrix $\mathbf{J}_f$ of $\mathbf{f}$ with respect to $\mathbf{y}$ at a given point.

9.2.3 Implicit methods (Euler backward method)

Euler's method is an **explicit method** in that it uses only information at time t_k to advance the solution to time t_{k+1} .

Methods which also use information at time t_{k+1} are called **implicit methods**.

The simplest example is the **Euler backward** method

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(t_{k+1}, \mathbf{y}_{k+1})$$

> **Derivation**
>
> This method can again be derived using a Taylor expansion:
>
> Consider the Taylor series
>
> $$\mathbf{y}(t-h) = \mathbf{y}(t) - h\mathbf{y}'(t) + \frac{1}{2}h^2\mathbf{y}''(t) + \cdots$$
>
> By taking $t = t_{k+1}$, $h = h_k$, $\mathbf{y}'(t_{k+1}) = \mathbf{f}(t_{k+1}, \mathbf{y}_{k+1})$, and dropping terms of second and higher order, we find:
>
> $$\mathbf{y}(t_{k+1} - h) = \mathbf{y}(t_{k+1}) - h_k \mathbf{f}(t_{k+1}, \mathbf{y}_{k+1})$$
>
> Because $t_{k+1} - h = t_k$, and by isolating $\mathbf{y}(t_{k+1})$, this is Euler's backward method.

The backward Euler method is implicit because we must evaluate f with the argument y_{k+1} *before we know its value*. This is not a problem because this statement simply means that a value for y_{k+1} that satisfies the preceding equation must be determined, and if f is a nonlinear function of y, as is often the case, then an iterative solution method, as seen in the nonlinear equations notebook, must be used.

A good starting guess for the iteration can be obtained from an explicit method, such as Euler's method, or from the solution at the previous time step.

Example

Consider the nonlinear scalar ODE $y' = -y^3$ with initial condition $y(0) = 1$.

Using the backward Euler method with a step size of $h = 0.5$, we obtain the equation

$$y_1 = y_0 + hf(t_1, y_1) = 1 - 0.5y_1^3$$

for the solution value at the next step.

You can solve this nonlinear equation using fixed-point-iteration with an initial guess given by the current value of $y_0 = 1$, or using an initial guess given by the solution of the Euler forward method, as is done below.

$$y_1 = y_0 - 0.5y_0^3 = 0.5$$

```
optimize.fixed_point((lambda y: 1.0 - 0.5 * y**3), 0.5, method="iteration")
```

```
np.float64(0.7709170003647279)
```

The analytical solution to this problem is

$$y = \frac{1}{\sqrt{2x+1}}$$

resulting in a value of $y(0.5) = 0.7071067811865475$

```
def analytical(x):
    return 1.0 / np.sqrt(2.0 * x + 1.0)


analytical(0.5)
```

```
np.float64(0.7071067811865475)
```

Note that both the Euler forward (0.5) and Euler backward (0.77) methods result in an estimate that is quite far from the real value, given the large step size of 0.5.

Repeating this procedure with a smaller step size results in better estimates.

```
def demo_stepsize(N):
    """Function that show the accuracy of the Euler forward
    and Euler backward functions to numerically integrate
    y'=-y^3 with y(0)=1 from 0 to 0.5
    as function of the step size taken in this interval

    Parameters
    ----------
        N
            A measure for the number of steps taken between 0 and 1.
            N=1 means that h=0.5,
            N=2 that h=0.25
            and so on.
```

(continues on next page)

(continued from previous page)

```
    """

    # analytical solution
    def func(x):
        return 1.0 / np.sqrt(2.0 * x + 1.0)

    # check with euler forward with smaller steps sizes of 0.5/N
    y = 1
    for _ in np.arange(N):
        y = y - 0.5 / N * y**3.0

    print("Euler forward result\n----------------------------")
    print("Step size, result, error")
    print(0.5 / N, y, np.abs(y - func(0.5)))

    # same for euler backward

    y = 1
    for _ in np.arange(N):
        yold = y
        y = optimize.fixed_point(
            (lambda ynew, yold=yold: yold - 0.5 / N * ynew**3),
            yold,
            method="iteration",
        )

    print("\nEuler backward result\n----------------------------")
    print("Step size, result, error")
    print(0.5 / N, y, np.abs(y - func(0.5)))


demo_stepsize(10)
```

```
Euler forward result
----------------------------
Step size, result, error
0.05 0.6974455049940009 0.00966127619254653

Euler backward result
----------------------------
Step size, result, error
0.05 0.7158805208572476 0.008773739670700165
```

Stability of the Euler backward method

Given the extra trouble and computation in using an implicit method, one might wonder why we would bother. The answer is that implicit methods generally have a significantly larger stability region than comparable explicit methods. To determine the stability and accuracy of the backward Euler method, we apply it to the scalar test ODE $y' = \lambda y$, obtaining

$$y_{k+1} = y_k + h\lambda y_{k+1}$$

or

$$(1 - h\lambda)y_{k+1} = y_k$$

so that

$$y_k = \left(\frac{1}{1-h\lambda}\right)^k y_0$$

Thus, for the backward Euler method to be stable we must have

$$\left|\frac{1}{1-h\lambda}\right| \leq 1$$

which holds for any $h > 0$ when $\mathrm{Re}(\lambda) < 0$. Thus, the stability region for the backward Euler method includes the entire left half of the complex plane, or the interval $(-\infty, 0)$ if λ is real, and there is no stability restriction on the step size when computing a stable solution. The growth factor

$$\frac{1}{1-h\lambda} = 1 + h\lambda + (h\lambda)^2 + \cdots$$

agrees with the expansion for $e^{h\lambda}$ through terms of order h, so the backward Euler method is first-order accurate.

For any ODE, the stability region for the backward Euler method includes the entire left half of the complex plane, and hence for computing a stable solution, the method is stable for any positive step size. Such a method is said to be **unconditionally stable**. The great virtue of an unconditionally stable method is that the desired local accuracy places the only constraint on our choice of step size. Thus, we may be able to take much larger steps than for an explicit method of comparable order and attain much higher overall efficiency despite requiring more computation per step because of having to solve an equation at each step of the implicit method.

However, not all implicit methods have this property. Implicit methods generally have larger stability regions than explicit methods, but the allowable step size is not always unlimited. Implicitness alone is not sufficient to guarantee stability.

9.2.4 Stiffness

Stiffness is a concept that can be defined a number of ways. For us, the most meaningful way is its correspondence to the physics behind the problems we are investigating.

If a system contains dynamics on very different timescales, like a slow relaxation towards a certain equilibrium, but with rapid oscillations around it, or with very strongly damped transients, then it is considered stiff.

Mathematically, a stable ODE $\mathbf{y}_0 = \mathbf{f}(t, \mathbf{y})$ is stiff if its Jacobian matrix $\mathbf{J}_f$ has eigenvalues that differ greatly in magnitude. There may be eigenvalues with relatively large negative real parts (corresponding to strongly damped components of the solution) or relatively large imaginary parts (corresponding to rapidly oscillating components of the solution).

Some numerical methods are very inefficient for stiff equations because the rapidly varying component of the solution forces very small step sizes to be used to maintain stability. Since the stability restriction depends on the rapidly varying component of the solution, whereas the accuracy restriction depends on the slowly varying component, the step size may be much more severely restricted by stability than by the required accuracy.

Euler's forward method, for example, is extremely inefficient for solving a stiff equation because of its small stability region. The unconditional stability of the implicit backward Euler method, on the other hand, makes it suitable for stiff problems.

Stiff ODEs need not be difficult to solve numerically, provided a suitable method, generally implicit, is chosen. This is illustrated with an example below:

> **Example**
>
> Consider the stiff ODE
>
> $$y' = -100y + 100t + 101$$
>
> with $y(0) = 1$.
>
> The analytical solution to this equation (in general) is $1+t+ce^{-100t}$ (i.e. with a very strongly damped component).

For our specific initial value, $c = 0$, so the solution becomes the linear equation $y(t) = 1 + t$, which in principle would be very well suited to solve with Euler's forward method.

However, to illustrate the effect of truncation or rounding errors, let us perturb the initial value slightly.

With a step size h = 0.1, the first few steps for the given initial values are given for the analytical solution in the following table

t	0.0	0.1	0.2	0.3	0.4	0.5
y(t)	1	1.1	1.2	1.3	1.4	1.5

Let's compare this with the numerical solutions obtained with the Euler forward and backward methods.

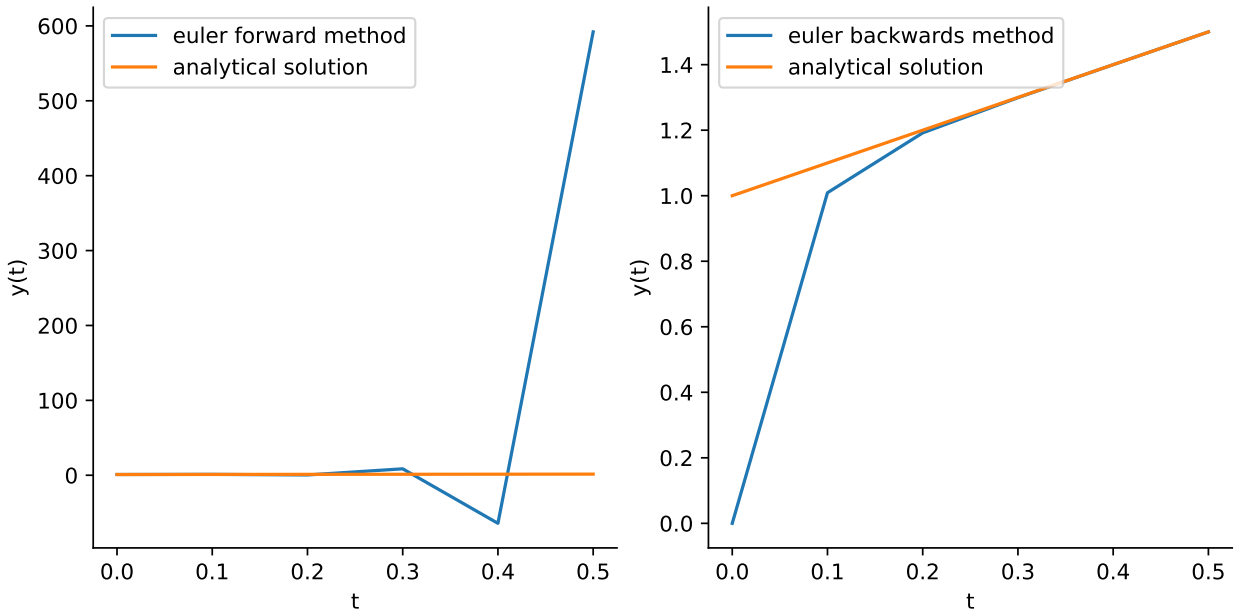
```
def demo_stiff():
    """Function that show the accuracy of the Euler forward
    and Euler backward functions to numerically integrate
    a very stiff problem

    The output loooks as follows:
    - the first column will contain the time points
    - the second column will contain the analytical results
    - the third column the approximation using euler forward
    - the fourth column the approximation using euler backward
    """

    # define empty array with correct dimension to store the output
    results = np.zeros((6, 4))

    # define 6 timepoints, 0,0.1,...0.5 and the analytical solution
    results[:, 0] = np.arange(0.0, 0.6, 0.1)
    results[:, 1] = results[:, 0] + 1

    # define function y'=-100y+100t+101
    def func(y, t):
        return -100 * y + 100 * t + 101

    # define helper function to print header
    def print_header():
        print("    ", "t,   analytical, forward, backward")

    # define helper function to print results
    def print_results():
        for i in np.arange(6):
            print(results[i, :])

    # euler forward method, starting from slightly perturbed initial value␣
↪y=0.99
    results[0, 2] = 0.99
    for i in np.arange(1, 6):
        results[i, 2] = results[i - 1, 2] + 0.1 * func(
            results[i - 1, 2], results[i - 1, 0]
        )

    # euler backwards method, starting from very perturbed initial value y=0.
```

(continues on next page)

(continued from previous page)

```
    results[0, 3] = 0.0
    for i in np.arange(1, 6):
        # Because f is linear in y, it's very easy to explicitly write
        # the analytical solution to  y1 = y0 + h*yf(t1,y1),
        # instead of using an iterative method y1 = [y0+h(100t1+101)] /␣
↪(101*h)
        results[i, 3] = (
            1.0 / 11.0 * (results[i - 1, 3] + 0.1 * (100.0 * results[i, 0] +␣
↪101.0))
        )

    print_header()
    with np.printoptions(precision=3, suppress=True):
        print(results)

    plt.close("stiff")
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4), num="stiff")
    ax1.plot(results[:, 0], results[:, 2], label="euler forward method")
    ax2.plot(results[:, 0], results[:, 3], label="euler backwards method")
    for ax in (ax1, ax2):
        ax.set(xlabel="t", ylabel="y(t)")
        ax.plot(results[:, 0], results[:, 1], label="analytical solution")
        ax.legend(loc="upper left")


demo_stiff()
```

```
     t,   analytical, forward, backward
[[  0.      1.      0.99    0.   ]
 [  0.1     1.1     1.19    1.009]
 [  0.2     1.2     0.39    1.192]
 [  0.3     1.3     8.59    1.299]
 [  0.4     1.4   -64.21    1.4  ]
 [  0.5     1.5   591.99    1.5  ]]
```



As you can see, even a tiny perturbation ruins the solution obtained using the Euler forward solver, whereas even a large perturbation eventually damps out as the result tends toward the correct solution using Euler backward.

9.2.5 Runge-Kutta methods

We can construct better methods, by not only relying on the first derivative of y', but also taking into account higher order derivatives. Such methods are called **Taylor series methods**. However such methods are only practical if the higher order derivatives of y are known analytically and are practical to calculate.

Runge-Kutta methods are single-step methods that are similar in motivation to Taylor series methods but do not involve explicit computation of higher derivatives. Instead, Runge-Kutta methods replace higher derivatives by finite difference approximations based on values of f at points between t_k and t_{k+1}.

This requires some bootstrapping to obtain the necessary values of f, since we do not know the second argument of f, namely the solution $y(t)$, for t between t_k and t_{k+1}.

The best-known Runge-Kutta method is the classical fourth-order scheme

$$\mathbf{y}_{k+1} = \mathbf{y}_k + \frac{h_k}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4)$$

with

$$\begin{aligned}
\mathbf{k}_1 &= \mathbf{f}(t_k, \mathbf{y}_k) \\
\mathbf{k}_2 &= \mathbf{f}(t_k + h_k/2, \mathbf{y}_k + (h_k/2)\mathbf{k}_1) \\
\mathbf{k}_3 &= \mathbf{f}(t_k + h_k/2, \mathbf{y}_k + (h_k/2)\mathbf{k}_2) \\
\mathbf{k}_4 &= \mathbf{f}(t_k + h_k, \mathbf{y}_k + h_k\mathbf{k}_3)
\end{aligned}$$

Runge-Kutta methods have a number of virtues.

- To proceed to time t_{k+1}, they require no history of the solution prior to time t_k
  - which makes them self-starting at the beginning of the integration
  - and also makes it easy to change the step size during the integration
- These features also make Runge-Kutta methods relatively easy to program, which accounts in part for their popularity.

Butcher tableaus

The notation to describe these solvers quickly becomes quite dense and difficult to read. Butcher invented a method to write down all necessary information to understand and implement a solver in a very elegant way: **Butcher tableaus**.

In general, the approximate solution of the ODE

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

at time $t + h$, given its value y on time t, is obtained by taking s intermediate evaluations of f at times c_i,

$$k_i = f\left(t + c_i h, y_n + h \sum_{j=1}^{s} a_{ij} k_j\right)$$

and then adding them to y_n with the correct weights b_i:

$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i k_i$$

These formulas can compactly be represented by a Butcher tableau as follows:

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \cdots & a_{1s} \\ c_2 & a_{21} & a_{22} & \cdots & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\ \hline & b_1 & b_2 & \cdots & b_s \end{array}$$

If the tableau only contains elements below the diagonal, then it corresponds to an **explicit** solver. Otherwise it is an **implicit** solver.

Below, the Butcher tableaus of all solvers mentioned above are given. For the methods which have a lower order solution embedded, the lower order solution is shown as an extra row below the higher order solution. The difference between both solutions approximates the error.

- **Euler Forward**

$$\begin{array}{c|c} 0 & \\ \hline & 1 \end{array}$$

- **Euler Backward**

$$\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array}$$

- **4th order Runke-Kutta solver**

$$\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$$

Example derivation (2nd order Runge-Kutta method)

We are trying to solve the differential equation

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

with second-order accuracy using what will turn out to be Heun's method.

The Butcher tableau for this method looks like:

$$\begin{array}{c|cc} 0 & & \\ c_1 & a_{21} & \\ \hline & b_1 & b_2 \end{array}$$

We'll introduce shortcuts for notational clarity:

- $f(t, y) = f$
- Partial derivatives are written as a subscript:
  - $\frac{\partial f}{\partial t} = f_t$
  - $\frac{\partial f}{\partial y} = f_y$

We use the general form for a second-order Runge-Kutta method:

$$k_1 = f(t, y_n)$$

$$k_2 = f(t + c_1 h, y_n + a_{21} h k_1)$$

By taking the second-order Taylor expansion for k_1 and k_2:

$$k_1 = f(t, y_n) = f$$

$$k_2 = f(t + c_1 h, y_n + a_{21} h k_1) = f + c_1 h f_t + a_{21} h f f_y + \frac{c_1^2 h^2}{2} f_{tt} + a_{21}^2 \frac{h^2}{2} f^2 f_{yy} + a_{21} c_1 h^2 f f_{ty}$$

We now substitute into the general formula for updating (y):

$$y_{n+1} = y_n + h(b_1 k_1 + b_2 k_2)$$

This becomes:

$$\begin{aligned} y_{n+1} = y_n + b_1 h f + b_2 h f + b_2 c_1 h^2 f_t + b_2 a_{21} h^2 f f_y \\ + b_2 \frac{c_1^2 h^3}{2} f_{tt} + b_2 a_{21} c_1 h^3 f f_{ty} + b_2 \frac{a_{21}^2 h^3}{2} f^2 f_{yy} \end{aligned}$$

We can now compare this to the Taylor expansion of $y(t)$ up to second order:

$$y_{n+1} = y_n + hf + \frac{h^2}{2}(f_t + f_y f)$$

From this comparison, we obtain the following system of equations to determine the unknowns b_1, b_2, c_1, and a_{21}:

$$\begin{aligned} b_1 + b_2 &= 1 \\ b_2 c_1 &= \frac{1}{2} \\ b_2 a_{21} &= \frac{1}{2} \end{aligned}$$

Rewriting these equations:

$$\begin{aligned} b_1 &= 1 - b_2 \\ b_2 &= \frac{1}{2c_1} \\ a_{21} &= c_1 \end{aligned}$$

Finally, by choosing (c_1 = 1), we get the **Heun method** with the following Butcher tableau:

$$\begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

This formulation shows the Heun method, where $c_1 = 1$, $a_{21} = 1$, and $b_1 = b_2 = \frac{1}{2}$.

Example derivation (3rd order Runge-Kutta method)

As another example we will derive our very own explicit third order solver.

We are trying to solve a first order differential equation

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

with third order accuracy.

The Butcher tableau we are trying to fill in looks like this:

$$\begin{array}{c|ccc} 0 & & & \\ a & c & & \\ b & d & e & \\ \hline & F & G & H \end{array}$$

As already mentioned previously, the notation quickly becomes very dense. Therefore we will introduce some notational shortcuts:

- Whenever we evaluate f at (t, y), we no longer write its argument: $f(t, y) = f$
- Partial derivatives are written as a subscript:
  - $\frac{\partial f}{\partial t} = f_t$
  - $\frac{\partial f}{\partial y} = f_y$

Using

$$k_i = f\left(t + c_i h, y_n + h \sum_{j=1}^{s} a_{ij} k_j\right)$$

and each time taking the second order Taylor expansion we can find k_1,k_2 and k_3 up to order h^2 (after substituting k_1 and k_2 in k_2 and k_3 where necessary):

$$\begin{aligned}
k_1 &= f(t, y_n) = f \\
k_2 &= f(t + ah, y_n + hck_1) \\
&= f + ahf_t + chff_y + a^2\frac{h^2}{2}f_{tt} + ach^2 ff_{ty} + c^2\frac{h^2}{2}f^2 f_{yy} \\
k_3 &= f(t + bh, y_n + hdk_1 + hek_2) \\
&= f + bhf_t + dhff_y + ehff_y + aeh^2 f_y f_t + ceh^2 ff_y^2 \\
&\quad + b^2\frac{h^2}{2}f_{tt} + bdh^2 ff_{ty} + beh^2 ff_{ty} + d^2\frac{h^2}{2}f^2 f_{yy} \\
&\quad + e^2\frac{h^2}{2}f^2 f_{yy} + deh^2 f^2 f_{yy}
\end{aligned}$$

inserting this in

$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i k_i$$

gives

$$\begin{aligned} y_{n+1} = y_n &+ Fhf + Ghf + Gah^2 f_t + Gch^2 f f_y + Ga^2 \frac{h^3}{2} f_{tt} \\ &+ Gach^3 f_{ty} + Gc^2 \frac{h^3}{2} f^2 f_{yy} + Hhf + Hbh^2 f_t + Hdh^2 f f_y \\ &+ Heh^2 f f_y + Haeh^3 f_y f_t + Hceh^3 f f_y^2 + Hb^2 \frac{h^3}{2} f_{tt} \\ &+ Hbdh^3 f f_{ty} + Hbeh^3 f f_{ty} + Hd^2 \frac{h^3}{2} f^2 f_{yy} \\ &+ He^2 \frac{h^3}{2} f^2 f_{yy} + Hdeh^3 f^2 f_{yy}, \end{aligned}$$

which we can compare with the Taylor expansion up to order 3:

$$y_{n+1} = y_n + hf + \frac{h^2}{2}(f_t + f_y f) + \frac{h^3}{6}(f_{tt} + 2f_{ty} f + f_t f_y + f^2 f_{yy} + f_y^2 f)$$

to find the following set of equations to determine the unknowns a up to G:

$$\begin{aligned} F + G + H &= 1 \\ aG + bH &= 1/2 \\ cG + H(d+e) &= 1/2 \\ Ga^2 + Hb^2 &= 1/3 \\ Gac + Hb(d+e) &= 1/3 \\ Hea &= 1/6 \\ Ga^2 + H(d+e)^2 &= 1/3 \\ Hec &= 1/6 \end{aligned}$$

When defining a and b ourselves, this can be written as:

$$\begin{aligned} a &= a \\ b &= b \\ c &= a \\ d &= \frac{b(3a^2 - 3a + b)}{a(3a-2)} \\ e &= b - d \\ F &= \frac{6ab - 3a - 3b + 2}{6ab} \\ G &= \frac{3b-2}{6a(b-a)} \\ H &= \frac{3a-2}{6b(a-b)} \end{aligned}$$

When choosing $a = 1/2$ and $b = 1$, we obtain the **third order Runge-Kutta method**

$$\begin{array}{c|ccc} 0 & & & \\ \frac{1}{2} & \frac{1}{2} & & \\ 1 & -1 & 2 & \\ \hline & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{array}$$

However, if we would like to find our own solver, we choose any value we like for a and b.

Adaptive step size

Classical Runge-Kutta methods provide no error estimate on which to base the choice of step size and therefore require a **fixed step**.

When a solver contains not only a solution of order $\mathcal{O}(N)$ but also a solution of $\mathcal{O}(N-1)$, we can use the difference between both solutions as an approximation of the error size ϵ on the solution. This error depends on the size of the time step h, and given a certain error tolerance τ, it is possible to suggest a h for the next time step which is a large as possible, while still maintaining the level of accuracy required as follows:

$$h_{\text{optimal}} = h_{\text{current}} \left(\frac{\tau}{\epsilon} \right)^{(1/N)}$$

These solvers can therefore use an **adaptive step size**.

In systems governed by dynamics whose speed changes in time, this can boost the performance of a solver tremendously. Even if the *adaptive step* does not need to adapt itself a lot and remains more or less constant, you have the advantage that your simulation runs at the best possible efficiency, given the required accuracy.

A simple example of such a solver is **Heun's method**. In this method, the first order accurate solution is found using Euler's forward method

$$\tilde{y}_{n+1} = y_n + h f(t, y_n)$$

Afterwards, the second order solution is found:

$$y_{n+1} = y_n + \frac{h}{2} \left(f(t, y_n) + f(t + h, \tilde{y}_{n+1}) \right)$$

The difference between both solutions is an estimate of the error.

The butcher tableau looks like this:

$$\begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & \frac{1}{2} & \frac{1}{2} \\ & 1 & 0 \end{array}$$

Probably the most used embedded pair method was developed by Dormand and Prince. This is a 5th order accurate solver with a 4th order embedded error estimate. This solver is the default solver in `scipy` and in matlab.

Its butcher tableau is given here

- **Dormand-Prince method**

$$\begin{array}{c|ccccccc} 0 & & & & & & & \\ \frac{1}{5} & \frac{1}{5} & & & & & & \\ \frac{3}{10} & \frac{3}{40} & \frac{9}{40} & & & & & \\ \frac{4}{5} & \frac{44}{45} & -\frac{56}{15} & \frac{32}{9} & & & & \\ \frac{8}{9} & \frac{19372}{6561} & -\frac{25360}{2187} & \frac{64448}{6561} & -\frac{212}{729} & & & \\ 1 & \frac{9017}{3168} & -\frac{355}{33} & \frac{46732}{5247} & \frac{49}{176} & -\frac{5103}{18656} & & \\ 1 & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & -\frac{2187}{6784} & \frac{11}{84} & \\ \hline & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & -\frac{2187}{6784} & \frac{11}{84} & 0 \\ & \frac{5179}{57600} & 0 & \frac{7571}{16695} & \frac{393}{640} & -\frac{9209}{339200} & \frac{187}{2100} & \frac{1}{40} \end{array}$$

First-same-as-last (FSAL) property

When we have a closer look at Heun's method, we see that we need two evaluations per time step.

Now consider the **Bogacki-Shampine** method, which is a third order method with embedded second order solution. This method seems to require 4 function evaluations per step.

$$
\begin{array}{c|cccc}
0 & & & & \\
\frac{1}{2} & \frac{1}{2} & & & \\
\frac{3}{4} & 0 & \frac{3}{4} & & \\
1 & \frac{2}{9} & \frac{1}{3} & \frac{4}{9} & \\
\hline
 & \frac{2}{9} & \frac{1}{3} & \frac{4}{9} & 0 \\
 & \frac{7}{24} & \frac{1}{4} & \frac{1}{3} & \frac{1}{3}
\end{array}
$$

However, contrary to Heun's method, in the Bogacki-Shampine method the last evaluation of step n corresponds with the first evaluation of step $n + 1$ (which is shown as the identical lines in bold), thus effectively reducing the number of evaluations per step to 3.

This property is called is the **first-same-as-last (FSAL)** property and makes this solver 4/3 times more efficient compared to the case when it didn't have the FSAL property.

Performance

In this section we will validate the implementation of the different solvers by investigating the error on the solution as a function of the step size. If the solvers are implemented correctly, this error should go down to the level of numerical noise at low time steps at a rate proportional to h^N for a solver of order $\mathcal{O}(N)$.

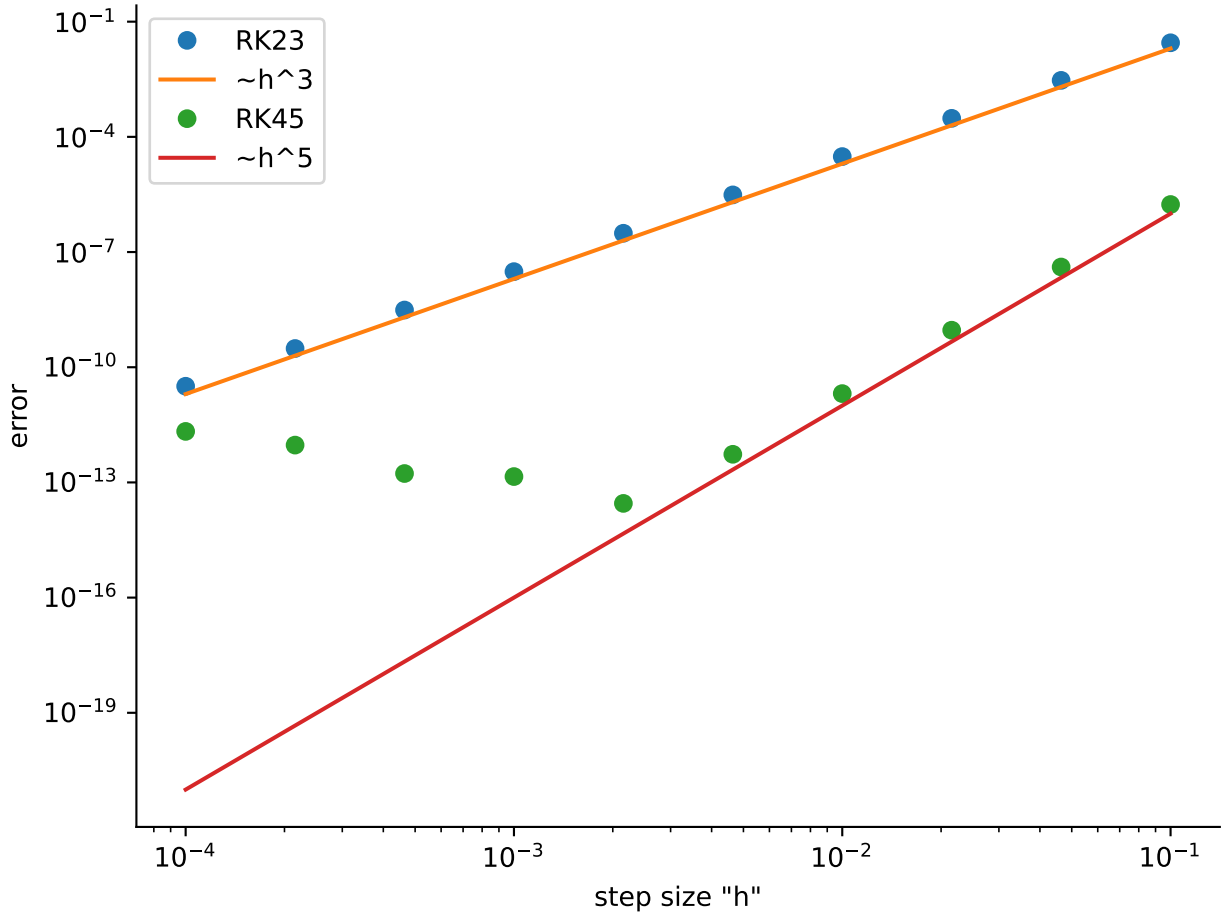
> **Example**
>
> As an example we'll integrate the ODE $y'(t) = y(t)$ with $y(0) = 1$ from $t = 0$ to $t = 5$ with a variable step size h between 10^{-4} and 10^{-1}, and investigate the error on the result at $t = 5$ as function of h.

```
def performance_demo():
    """Demonstrate the obtained error in the solution of an example ODE y'=y
    as function of step size for a 3rd and 5th order solver.
    """

    # Define ODE
    def ODE(t, y):
        return y

    # empty table to store perfomance for 10 values of h
    # the first column contains h
    # the second column error on the result obtained with RK23
    # the third column error on the result obtained with R45
    perf = np.zeros([10, 3])

    steps = np.logspace(-4, -1, num=10)
    # solve ODE
    for i, h in enumerate(steps):
        sol = integrate.solve_ivp(ODE, [0, 5], [1], max_step=h, method="RK23")
        sol2 = integrate.solve_ivp(ODE, [0, 5], [1], max_step=h, method="RK45
↪")
        perf[i] = [
            h,
            np.abs(sol.y[0, -1] - np.exp(5)),
            np.abs(sol2.y[0, -1] - np.exp(5)),
        ]
```

(continues on next page)

(continued from previous page)

```
    # plot results
    plt.close("performance")
    fig, ax = plt.subplots(num="performance")
    ax.plot(perf[:, 0], perf[:, 1], "o", label="RK23")
    ax.plot(perf[:, 0], 20 * perf[:, 0] ** 3.0, label="~h^3")
    ax.plot(perf[:, 0], perf[:, 2], "o", label="RK45")
    ax.plot(perf[:, 0], 0.1 * perf[:, 0] ** 5.0, label="~h^5")
    ax.set_xscale("log")
    ax.set_yscale("log")
    ax.set_xlabel('step size "h"')
    ax.set_ylabel("error")
    ax.legend()


performance_demo()
```



As seen in the figure, the error indeed scales as the step size to the third and fifth power for the third and fifth order solver, respectively. Note that, as the solver reaches the machine precision of about 10^{-16}, it is of no use to further decrease the step size.

Example

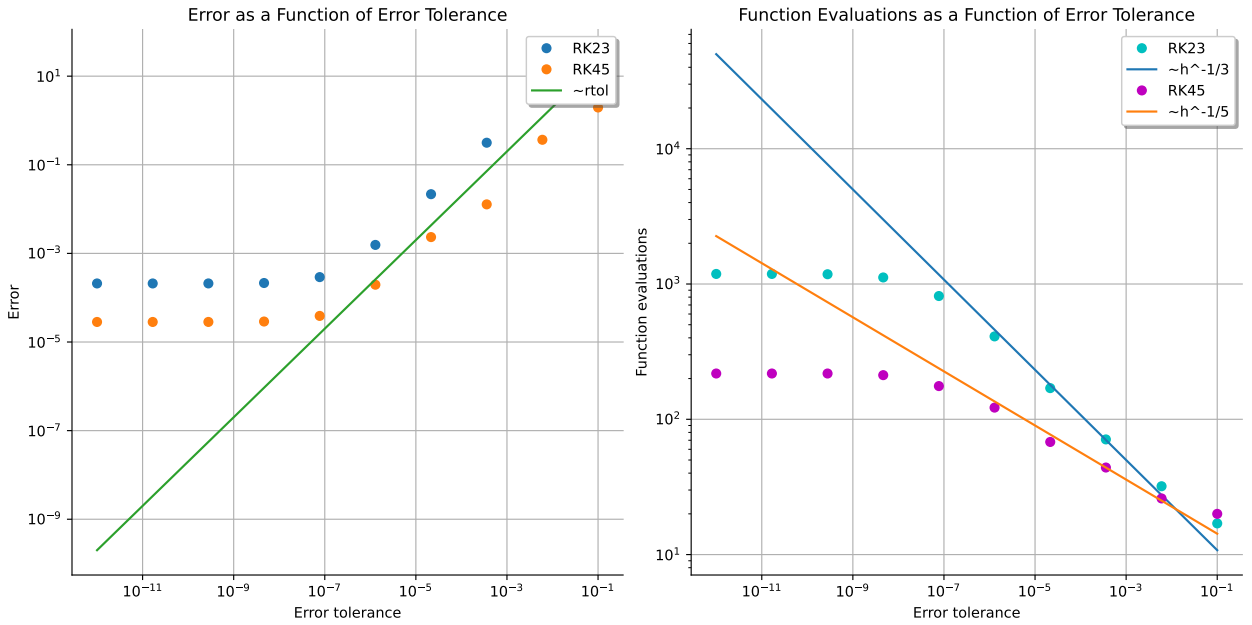
Secondly, we'll also have a look at the obtained error and number of function evaluations taken as function of a predefined relative error tolerance for 2 adaptive step size solvers.

```
def performance_demo2():
    """Demonstrate the obtained error and number of function evaluations taken
    as function of the used relative error when solving of an example ODE y'=y
    with an adaptive step 3rd and 5th order solver.
    """

    # Define ODE
    def ODE(t, y):
        return y

    # Empty table to store performance for 10 values of the relative error␣
↪tolerance
    # First column: rtol
    # Second column: error on result obtained with RK23
    # Third column: number of function evaluations taken by RK23
    # Fourth column: error on result obtained with RK45
    # Fifth column: number of function evaluations taken by RK45
    perf = np.zeros([10, 5])

    # Solve ODE
    steps = np.logspace(-12, -1, num=10)
    for i, tol in enumerate(steps):
        sol = integrate.solve_ivp(ODE, [0, 5], [1], rtol=tol, method="RK23")
        sol2 = integrate.solve_ivp(ODE, [0, 5], [1], rtol=tol, method="RK45")
        perf[i] = [
            tol,
            np.abs(sol.y[0, -1] - np.exp(5)),
            sol.nfev,
            np.abs(sol2.y[0, -1] - np.exp(5)),
            sol2.nfev,
        ]

    # Plot results: Error as a function of error tolerance
    plt.close("performance_demo2")
    fig, (ax0, ax1) = plt.subplots(1, 2, squeeze=True, figsize=(12, 6), num=
↪"performance_demo2")
    ax0.set_xscale("log")
    ax0.set_yscale("log")
    ax0.plot(perf[:, 0], perf[:, 1], "o", label="RK23")
    ax0.plot(perf[:, 0], perf[:, 3], "o", label="RK45")
    ax0.plot(perf[:, 0], 200 * perf[:, 0], label="~rtol")
    ax0.set_xlabel("Error tolerance")
    ax0.set_ylabel("Error")
    ax0.legend(loc="upper right", fancybox=True, shadow=True)
    ax0.set_title("Error as a Function of Error Tolerance")
    ax0.grid(True)

    # Plot results: Number of function evaluations as a function of error␣
↪tolerance
```

(continues on next page)

(continued from previous page)

```
    ax1.set_xscale("log")
    ax1.set_yscale("log")
    ax1.plot(perf[:, 0], perf[:, 2], "o", c="c", label="RK23")
    ax1.plot(perf[:, 0], 5 * perf[:, 0] ** (-1.0 / 3), label="~h^-1/3")
    ax1.plot(perf[:, 0], perf[:, 4], "o", c="m", label="RK45")
    ax1.plot(perf[:, 0], 9 * perf[:, 0] ** (-1.0 / 5), label="~h^-1/5")
    ax1.set_xlabel("Error tolerance")
    ax1.set_ylabel("Function evaluations")
    ax1.legend(loc="upper right", fancybox=True, shadow=True)
    ax1.set_title("Function Evaluations as a Function of Error Tolerance")
    ax1.grid(True)

    plt.show()


performance_demo2()
```



At first sight, it seems to pay off to implement increasingly complex and higher-order solvers. However, for each additional order, a number of extra evaluations per step are necessary, as shown in the table below.

solver	Euler	Heun12	RK3	Bogacki – Shampine23	RK4	Dormand – Prince45	Fehlberg6	Fehlberg7
$\frac{\#\text{evaluations}}{\text{step}}$	1	2	3	4	4	6	8	13

There also exists a theoretical limit which order can be achieved by a certain number of evaluations per step.

$\mathcal{O}$(solver)	1	2	3	4	5	6	7	8
$\frac{\#\text{evaluations}}{\text{step}}$	1	2	3	4	6	7	9	11

Note that the Bogacki-Shampine (when not considering FSAL) and the Fehlberg methods appear to be suboptimal, but this stems from the fact that they also have a lower order solution embedded, which further increases the number of conditions their numbers in the Butcher tableau have to fulfill, and consequently require more variables and thus more evaluations per step.

A second point to take into account is the memory usage of these solvers. The Seventh order Fehlberg method is only slightly faster than the Sixth order Fehlberg method, but uses 13 evaluations per step, as compared to 8. It thus requires

almost twice the amount of memory. Especially in GPU-software, where the memory bandwidth often is a limiting factor, such considerations need to be taken into account.

9.2.6 Extrapolation methods

Extrapolation methods are based on the use of a single-step method to integrate the ODE over a given interval, $t_k \leq t \leq t_{k+1}$, using several different step sizes h_i and yielding results denoted by $\mathbf{Y}(h_i)$. This gives a discrete approximation to a function $Y(h)$, where $\mathbf{Y}(0) = \mathbf{y}(t_{k+1})$.

An interpolating polynomial or rational function $\hat{\mathbf{Y}}(h)$ is fit to these data, and $\hat{\mathbf{Y}}(0)$ is then taken as the approximation to $\mathbf{Y}(0)$.

We saw an example of this approach in **Richardson extrapolation** for numerical differentiation and integration.

Extrapolation methods are capable of achieving very high accuracy, but they tend to be much less efficient and less flexible than other methods for ODEs, so they are used mainly when extremely high accuracy is required and cost is not a significant factor.

9.2.7 Multistep methods

Whereas Runge-Kutta methods only use information of one previous point (i.e. a single step method), **Multistep methods** use information at more than one previous point to estimate the solution at the next point.

One of the most popular *explicit* multistep methods is the fourth-order **Adams-Bashforth** method, which uses information of 3 previous time steps, next to the current one:

$$\mathbf{y}_{k+1} = \mathbf{y}_k + \frac{h}{24}\left(55\mathbf{y}'_k - 59\mathbf{y}'_{k-1} + 37\mathbf{y}'_{k-2} - 9\mathbf{y}'_{k-3}\right)$$

Derivation

This derivation considers a scalar function y, but the results can be applied componentwise to nonscalar functions as well.

We derive a multistep method of the form

$$\mathbf{y}_{k+1} = \alpha\mathbf{y}_k + h(\beta_0\mathbf{y}'_k + \beta_1\mathbf{y}'_{k-1} + \beta_2\mathbf{y}'_{k-2} + \beta_3\mathbf{y}'_{k-3})$$

To determine the 5 coefficients $\alpha, \beta_0, \beta_1, \beta_2$ and β_3, we require that this formula exactly integrates the first 5 monomials $1, t, t^2, t^3$ and t^4.

$$\begin{aligned}
1 &= \alpha + h(\beta_0 \cdot 0 + \beta_1 \cdot 0 + \beta_2 \cdot 0 + \beta_3 \cdot 0) \\
t_{k+1} &= \alpha t_k + h(\beta_0 \cdot 1 + \beta_1 \cdot 1 + \beta_2 \cdot 1 + \beta_3 \cdot 1) \\
t^2_{k+1} &= \alpha t^2_k + h(\beta_0 2t_k + \beta_1 2t_{k-1} + \beta_2 2t_{k-2} + \beta_3 2t_{k-3}) \\
t^3_{k+1} &= \alpha t^3_k + h(\beta_0 3t^2_k + \beta_1 3t^2_{k-1} + \beta_2 3t^2_{k-2} + \beta_3 3t^2_{k-3}) \\
t^4_{k+1} &= \alpha t^4_k + h(\beta_0 4t^3_k + \beta_1 4t^3_{k-1} + \beta_2 4t^3_{k-2} + \beta_3 4t^3_{k-3})
\end{aligned}$$

Because this method needs to work for *any* value of t_k and h, we can conventiently choosing $t_k = 0$ and $h = 1$.

It then follows that $t_{k+1} = 1$, $t_{k-1} = -1$, $t_{k-2} = -2$ and , $t_{k-3} = -3$

The first equation in the system thus becomes

$$1 = \alpha \cdot 1 + h(0)$$

From which it follows that $\alpha = 1$. The remaining system of equations thus reduces to

$$
\begin{aligned}
1 &= \beta_0 + \beta_1 + \beta_2 + \beta_3 \\
1 &= -2\beta_1 - 4\beta_2 - 6\beta_3 \\
1 &= 3\beta_1 + 12\beta_2 + 27\beta_3 \\
1 &= 1 - 4\beta_1 - 32\beta_2 - 108\beta_3
\end{aligned}
$$

Which we can solve using `linalg.solve` (as seen in the linear systems notebook) to find the coefficients of the Adams-Bashforth method.

```
def solve_adams_bashforth():
    """Solve the linear system encountered in the derivation of
    the Adams-Bashforth method.

    It's solution returns the coefficients of said method.
    """
    A = np.array([[1, 1, 1, 1], [0, -2, -4, -6], [0, 3, 12, 27], [0, -4, -32,␣
↪-108]])
    b = np.array([1, 1, 1, 1])
    x = linalg.solve(A, b)

    with np.printoptions(
        formatter={"all": lambda x: str(fractions.Fraction(x).limit_
↪denominator())}
    ):
        print(x)


solve_adams_bashforth()
```

```
[55/24 -59/24 37/24 -3/8]
```

One of the most popular *implicit* multistep methods is the fourth-order **Adams-Moulton** method, which uses information of 2 previous time steps, next to the current one *and the next one*:

$$
\mathbf{y}_{k+1} = \mathbf{y}_k + \frac{h}{24}\left(9\mathbf{y}'_{k+1} + 19\mathbf{y}'_k - 5\mathbf{y}'_{k-1} + \mathbf{y}'_{k-2}\right)
$$

Exercise

A very similar derivation can be written down to find the coefficients of the Adams-Moulton method, which is left as an exercise.

Just like for single-step methods, implicit multistep methods are usually more accurate and stable than explicit multistep methods, but they require an initial guess to solve the resulting (usually nonlinear) equation for $\mathbf{y}_{k+1}$. A good initial guess is conveniently supplied by an explicit method, so the explicit and implicit methods can be used as a **predictor-corrector pair**. One of the most used pairs is the Adams-Bashforth predictor and Adams-Moulton corrector shown above.

One could use the corrector repeatedly (i.e., fixed-point iteration) until some convergence tolerance is met, but doing so may not be worth the expense. Instead, typically, a fixed number of corrector steps, often only one, is, giving a **PECE (predict, evaluate, correct, evaluate)** scheme. Although it has no effect on the value of $\mathbf{y}_{k+1}$, the second evaluation of $\mathbf{f}$ in a PECE scheme yields an improved value of $\mathbf{y}'_{k+1}$ for use in later steps.

A few properties of multistep methods worth knowing

- Since multistep methods require several previous solution values and derivative values, how do we get started initially, before we have any past history to use? One strategy is to use a single-step method, which requires no past history, to generate solution values at enough points to begin using a multistep method.
- Changing step size is complicated, since the interpolation formulas are most conveniently based on equally spaced intervals for several consecutive points, so multistep methods are not idealy suited for adaptive step sizes.
- A good local error estimate can be determined from the difference between the predictor and the corrector.
- Implicit methods have a much greater region of stability than explicit methods but must be iterated to convergence to realize this benefit fully (e.g., a PECE scheme is actually explicit, albeit in a somewhat complicated way).
- A properly designed implicit multistep method can be very effective for solving stiff equations.

9.2.8 Multivalue methods

Changing step size is difficult with multistep methods because the past history of the solution is most easily maintained at equally spaced intervals. Similar to multistep methods, **multivalue** methods are based on polynomial interpolation, but they avoid many of the implementation difficulties associated with multistep methods.

The main idea motivating multivalue methods is the observation that the interpolating polynomial itself can be evaluated at any point, not just at equally spaced intervals. The equal spacing associated with multistep methods is simply an artifact of the way the methods are represented as a linear combination of successive solution and derivative values with fixed weights.

Multivalue methods are therefore a direct extension of multistep methods which allow adaptive step sizes (at the cost of more function evaluations per step and a more complicated implementation).

9.2.9 Methods to solve ODE's with `scipy`

All documentation on solving initial value problems for ODE's in `scipy` can be found here:

https://docs.scipy.org/doc/scipy/tutorial/integrate.html#ordinary-differential-equations-solve-ivp

The main functions you can use are `integrate.solve_ivp` and `integrate.odeint`.

Be aware that the latter has a different (and older) API than the former, but are still commonly used.

> They wrap older solvers implemented in Fortran (mostly ODEPACK). While the interface to them is not particularly convenient and certain features are missing compared to the new API, the solvers themselves are of good quality and work fast as compiled Fortran code. In some cases, it might be worth using this old API.

The application of the LSODA vs RK45 methods to a stiff problem is illustrated in the example below.

```
def demo_scipy():
    def func(t, y):
        return -100 * y + 100 * t + 101

    y0 = np.array([10.99])
    solution = integrate.solve_ivp(
        func,
        [0, 5],
        y0,
        t_eval=np.arange(0, 5.1, 0.1),
        method="RK45",
        atol=1.49012e-8,
        rtol=1.49012e-8,
    )
    print("RK45\n", solution.nfev, "\n", solution.y)
```

(continues on next page)

(continued from previous page)

```
    solution = integrate.solve_ivp(
        func,
        [0, 5],
        y0,
        t_eval=np.arange(0, 5.1, 0.1),
        method="LSODA",
        atol=1.49012e-8,
        rtol=1.49012e-8,
    )
    print("LSODA\n", solution.nfev, "\n", solution.y)

    def func(y, t):
        return -100 * y + 100 * t + 101

    sol = integrate.odeint(func, y0, np.arange(0, 5.1, 0.1), full_
↪output=False)
    print("ODEINT (also LSODA)\n", sol.transpose())


demo_scipy()
```

```
RK45
 1394
 [[10.99        1.10045355  1.20000002  1.30000002  1.40000002  1.50000002
   1.60000003  1.70000002  1.80000004  1.90000002  2.00000005  2.10000001
   2.20000005  2.30000001  2.40000003  2.50000002  2.60000001  2.70000004
   2.79999999  2.89999999  3.00000001  3.10000002  3.19999998  3.29999999
   3.4         3.49999999  3.59999999  3.70000001  3.79999998  3.89999999
   3.99999999  4.09999995  4.19999997  4.29999998  4.39999998  4.49999995
   4.59999999  4.69999993  4.79999999  4.89999998  4.99999995  5.09999998
   5.19999999  5.29999998  5.39999996  5.5         5.59999997  5.69999999
   5.8         5.89999998  6.00000001]]
LSODA
 238
 [[10.99        1.10045354  1.20000002  1.3         1.4         1.5
   1.6         1.7         1.8         1.9         2.          2.1
   2.2         2.3         2.4         2.5         2.6         2.7
   2.8         2.9         3.          3.1         3.2         3.3
   3.4         3.5         3.6         3.7         3.8         3.9
   4.          4.1         4.2         4.3         4.4         4.5
   4.6         4.7         4.8         4.9         5.          5.1
   5.2         5.3         5.4         5.5         5.6         5.7
   5.8         5.9         6.        ]]
ODEINT (also LSODA)
 [[10.99        1.10045354  1.20000002  1.3         1.4         1.5
   1.6         1.7         1.8         1.9         2.          2.1
   2.2         2.3         2.4         2.5         2.6         2.7
   2.8         2.9         3.          3.1         3.2         3.3
   3.4         3.5         3.6         3.7         3.8         3.9
   4.          4.1         4.2         4.3         4.4         4.5
   4.6         4.7         4.8         4.9         5.          5.1
   5.2         5.3         5.4         5.5         5.6         5.7
   5.8         5.9         6.        ]]
```

Note that LSODA is much more accurate, while RK45 makes much more function evaluations.

9.3 Boundary Value Problems (BVP's) for ODE's

9.3.1 Introduction

By itself, a differential equation does not uniquely determine a solution; additional side conditions must be imposed on the solution to make it unique.

These side conditions prescribe values that the solution or its derivatives must have at some specified point or points. If all of the side conditions are specified at the same point, say t_0 , then we have an initial value problem, which we considered until now.

If the side conditions are specified at more than one point, then we have a **boundary value problem, or BVP**.

For an ordinary differential equation, the side conditions are typically specified at two points, namely the endpoints of some interval $[a, b]$, which is why the side conditions are called boundary conditions or boundary values.

The remainder of this notebook will introduce a numerical method for solving such **two-point boundary value problems**.

> **Examples**
>
> Newton's Second Law of Motion, $F = ma$, which is a second-order ODE, involves two constants of integration, and hence two side conditions must be specified in order to determine a unique solution over the interval of integration, say $[a, b]$.
>
> In an initial value problem, both the position $y(a)$ and velocity $y'(a)$ would be specified at the initial point a, and this would uniquely determine the solution $y(t)$ over the entire interval.
>
> Other side conditions could be specified, however, such as the initial position $y(a)$ and final position $y(b)$, or the initial position $y(a)$ and final velocity $y'(b)$. Indeed, any linear (or even nonlinear) combination of solution and derivative values at the endpoints could be specified, each giving a different two-point boundary value problem for this ODE.

> Our primary focus will be on second-order scalar BVPs (and equivalent first-order systems) of exactly this type because many important physical problems have this form, including
>
> - The bending of an elastic beam under a distributed transverse load
> - The distribution of electrical potential between two flat electrodes
> - The temperature distribution in an internally heated homogeneous wall whose surfaces are maintained at fixed temperatures
> - The steady-state concentration of a pollutant in porous soil

The two-point boundary value problem for the second-order scalar ODE

$$u'' = f(t, u, u')$$

with $a < t < b$

and boundary conditions:

- $u(a) = \alpha$
- $u(b) = \beta$

is equivalent to the first order system of ODEs:

$$\begin{bmatrix} y_1' \\ y_2' \end{bmatrix} = \begin{bmatrix} y_2 \\ f(t, (y1, y2) \end{bmatrix}$$

with $a < t < b$

and with separated boundary conditions

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} y_1(a) \\ y_2(a) \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} y_1(b) \\ y_2(b) \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

For the general first-order two-point boundary value problem

$$\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$$

with $a < t < b$ and boundary conditions

$$\mathbf{g}(\mathbf{y}(a), \mathbf{y}(b)) = \mathbf{0}$$

let $\mathbf{y}(t; \mathbf{x})$ denote the solution to the associated initial value problem with initial condition $\mathbf{y}(a) = x$. For a given $\mathbf{x}$, the solution $\mathbf{y}(t; \mathbf{x})$ of the IVP is a solution of the BVP if

$$\mathbf{h}(\mathbf{x}) \equiv \mathbf{g}(\mathbf{x}, \mathbf{y}(b; \mathbf{x})) = \mathbf{0}$$

9.3.2 Shooting method

The **shooting method** replaces a given boundary value problem by a sequence of initial value problems.

As indicate above, the general first-order two-point boundary value problem is equivalent to the system of nonlinear algebraic equations

$$\mathbf{h}(\mathbf{x}) \equiv \mathbf{g}(\mathbf{x}, \mathbf{y}(b; \mathbf{x})) = \mathbf{0}$$

One way to solve the BVP, therefore, is to solve the nonlinear system $\mathbf{h}(\mathbf{x}) = \mathbf{0}$ using any suitable method from the nonlinear systems notebook.

Evaluation of $\mathbf{h}(\mathbf{x})$ for any given value $\mathbf{x}$ will require solving an IVP to determine $\mathbf{y}(\mathbf{b}; \mathbf{x})$, for which we can use any suitable method seen in this notebook.

To make this approach more concrete, consider the two-point BVP for a scalar second-order ODE

$$u'' = f(t, u, u')$$

with $a < t < b$ and boundary conditions:

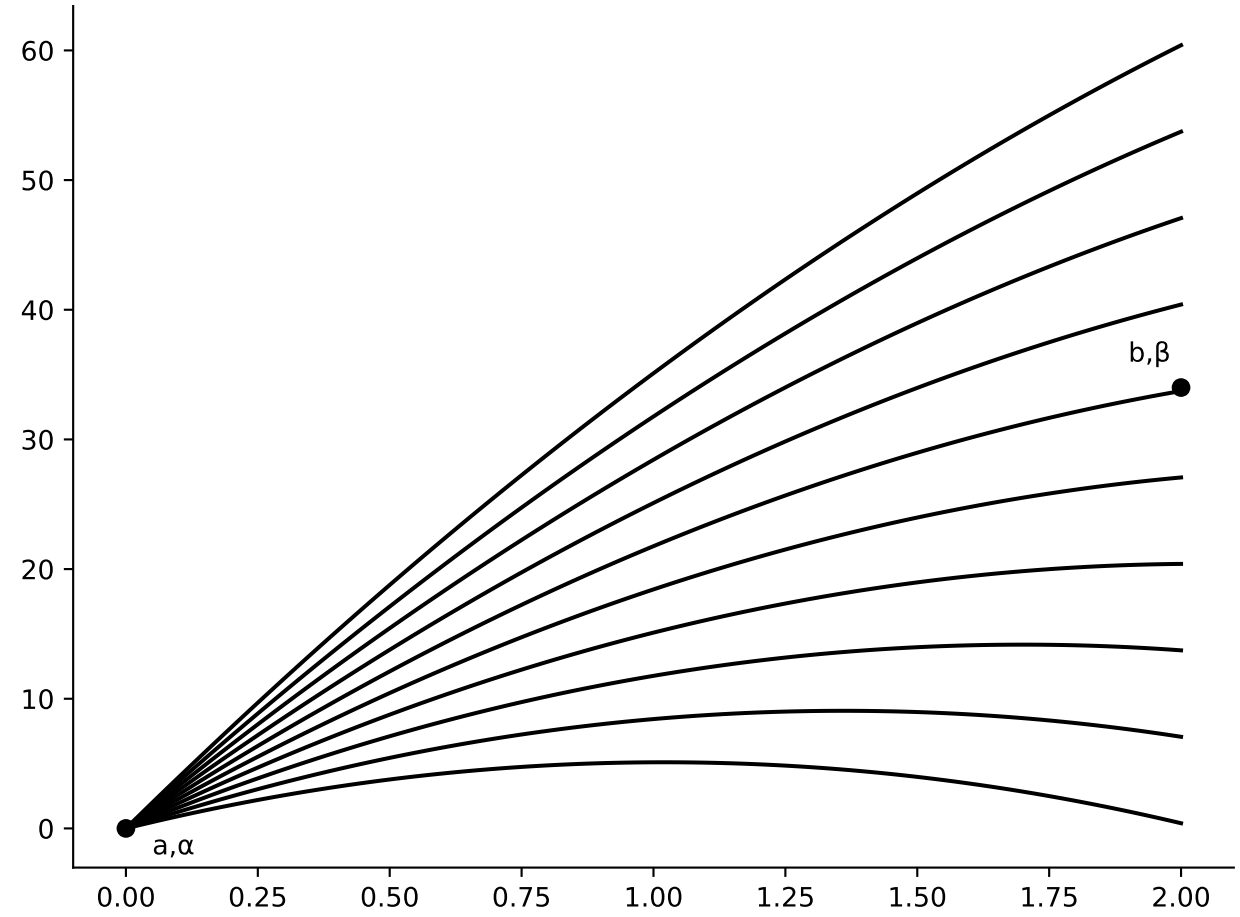
- $u(a) = \alpha$
- $u(b) = \beta$

where we are given the initial value $u(a)$ and final value $u(b)$ of the solution, but not the initial slope $u'(a)$. If we knew the latter, then we would have an IVP. We lack that information, but we can guess a value for the initial slope, solve the resulting IVP, and then check to see if the computed solution value at $t = b$ matches the desired boundary value, $u(b) = \beta$.

The basic idea is illustrated in the figure below.

Each curve represents a solution of the same second-order ODE, with different values for the initial slope $u'(a)$ giving different solutions for the ODE. All of the solutions start with the given initial value $u(a) = \alpha$, but for only one value of the initial slope does the resulting solution curve hit the desired boundary condition $u(b) = \beta$. The motivation for the name **shooting method** should now be obvious: we keep adjusting our aim until we hit the target.

```
def demo_shooting_method():
    """Visually illustrate the shooting method.

    ODE y''=-g

    or  y'=v
        v'=-g

    """

    def func(t, y):
        """Compute derivatives of y.

        Parameters
        ----------
        t
            time
        y
            Array with [position, velocity]

        Returns
        -------
        yprime
            Array with [velocity, accelaration]
        """
        return np.array([y[1], -9.8])

    y0 = 0
    solutions = np.zeros([10, 100])
    guesses = np.linspace(10, 40, 10)

    plt.close("shooting")
    fig, ax = plt.subplots(num="shooting")
    for i, v0 in enumerate(guesses):
        sol = integrate.solve_ivp(
            func, [0, 2], [y0, v0], t_eval=np.linspace(0, 2, 100)
        )
        solutions[i] = sol.y[0]
        ax.plot(sol.t, solutions[i, :], c="k")

    ax.plot(0, 0, "o", c="k")
    ax.annotate("a,α", (0.05, -2))
    ax.plot(2, 34, "o", c="k")
    ax.annotate("b,β", (1.9, 36))


demo_shooting_method()
```



After transforming a general second-order BVP into a first-order system, the boundary conditions become

$$\mathbf{g}(\mathbf{y}(a), \mathbf{y}(b)) = \begin{bmatrix} y_1(a) - \alpha \\ y_1(b) - \beta \end{bmatrix} = \mathbf{0}$$

Thus, the nonlinear system to be solved is

$$\mathbf{h}(\mathbf{x}) = \begin{bmatrix} y_1(a; \mathbf{x}) - \alpha \\ y_1(b; \mathbf{x}) - \beta \end{bmatrix} = \mathbf{0}$$

where x is the initial value. The first component of $\mathbf{h}(\mathbf{x})$ will be zero if $x_1 = \alpha$, and the initial slope x_2 remains to be determined so that the second component of $\mathbf{h}(\mathbf{x})$ will be zero.

In effect, therefore, we must solve the scalar nonlinear equation in x_2 ,

$$h_2(\alpha, x_2) = y_1(b; \alpha, x_2) = 0$$

for which we can use a one-dimensional zero finder seen in the nonlinear systems notebook.

> **Example**
>
> We illustrate the shooting method on the two-point BVP for the second-order scalar ODE which describes the vertical trajectory of a bullet (neglecting air friction).
>
> We shoot the bullet in the air at time and position equal to zero, and find that it takes 40 seconds for the bullet to fall down again.
>
> $$y'' = -g$$

with boundary conditions $y(0) = 0$ and $y(40) = 0$

For each guess for $y'(0)$, we will integrate the ODE using the Dormand-Prince method (the default for `integrate.solve_ivp` to determine how close we come to hitting the desired solution value at $t = 40$.

Before doing so, however, we must first transform the second-order ODE into a system of two first-order ODEs

$$\begin{bmatrix} y' \\ v' \end{bmatrix} = \begin{bmatrix} v \\ -g \end{bmatrix}$$

and write a function which gives us the resulting position at $t = 40$ for an initial guess $v(0)$.

and try an initial velocity of $v(0) = 100$ m/s.

```
def guess(v0):
    """Helper function to integrate the differential equation y''=-g
    from t=0 to t=40, given an initial value v0.
    """

    def func(t, y):
        return np.array([y[1], -9.8])

    # Fix the initial state, even when given as array,
    # for compatibility with root finder.
    y0 = 0
    v0 = np.ravel([v0])[0]
    return integrate.solve_ivp(func, [0, 40], [y0, v0], t_eval=[40]).y[0][0]


guess(100)
```

```
np.float64(-3839.9999999999955)
```

This has now become a root finding problem, which we can solve using `optimize.root`, with the initial guess of $v(0) = 100$.

```
optimize.root(guess, 100)
```

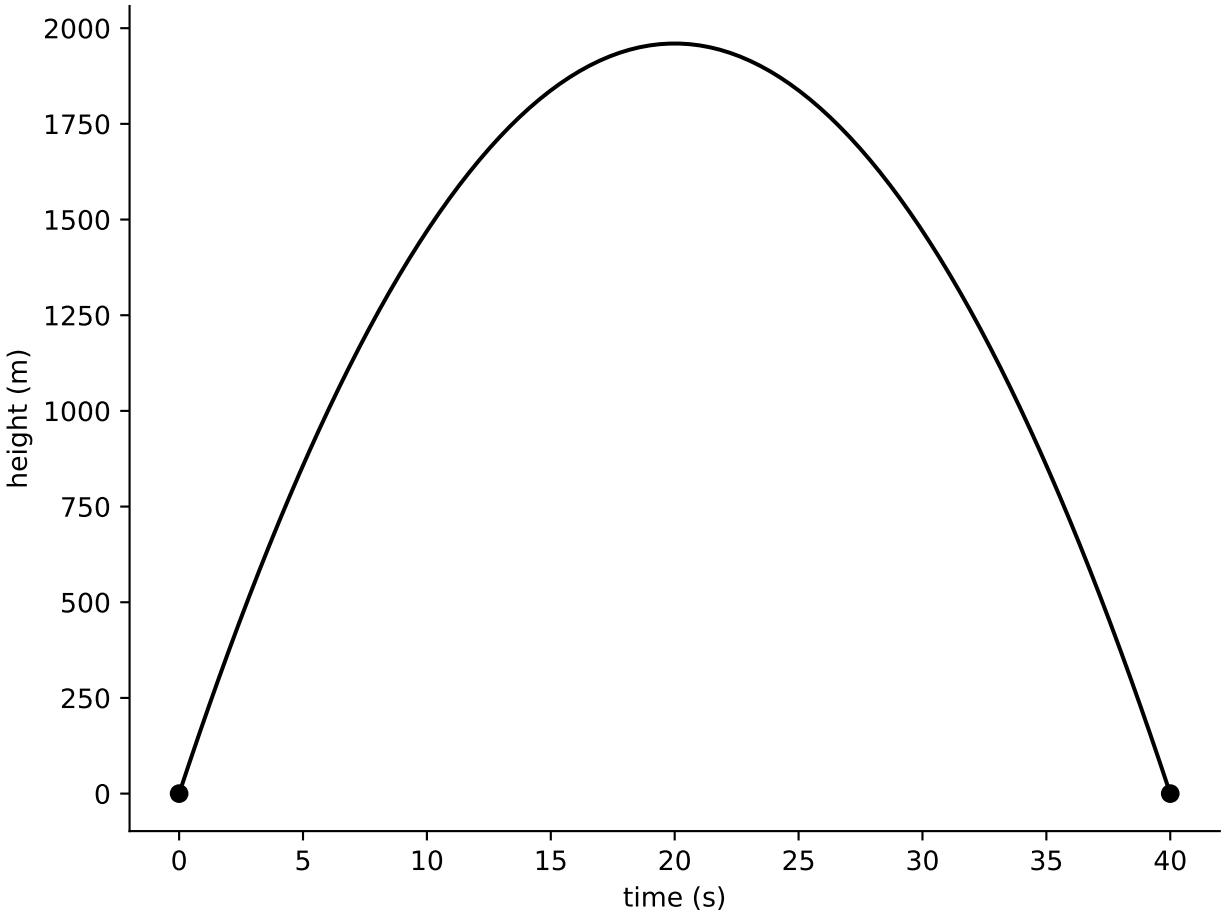
```
 message: The solution converged.
 success: True
  status: 1
     fun: -5.684341886080801e-13
       x: [ 1.960e+02]
  method: hybr
    nfev: 11
    fjac: [[-1.000e+00]]
       r: [-8.739e+01]
     qtf: [ 5.684e-13]
```

Resulting in the solution $v(0) = 196$ m/s.

> We also could have solved this problem immediately using `integrate.solve_bvp`

```
def demo_bvp():
    """Illustrate integrate.solve_bvp to solve the boundary value problem
    y''=-g with y(0)=y(40)=0 as boundary values.
    """

    def func(t, y):
        return np.array([y[1], -9.8 * np.ones(len(y[0]))])

    def bc(ya, yb):
        return np.array([ya[0], yb[0]])

    x = np.linspace(0, 40, 100)
    y = np.ones((2, 100))

    sol = integrate.solve_bvp(func, bc, x, y)

    print("v(0)= ", sol.y[1, 0])
    plt.close("bvp")
    fig, ax = plt.subplots(num="bvp")
    ax.plot(sol.x, sol.y[0], c="k")
    ax.plot(0, 0, "o", c="k")
    ax.plot(40, 0, "o", c="k")
    ax.set_xlabel("time (s)")
    ax.set_ylabel("height (m)")


demo_bvp()
```

```
v(0)=  196.00000000000003
```



The shooting method is conceptually simple and is easy to implement using existing software for initial value problems and for nonlinear equations. It has serious drawbacks, however.

Chief among these is that the shooting method inherits the stability (or instability) of the associated IVP, which as we have seen may be unstable even when the BVP is stable. This potential ill-conditioning of the IVP may make it extremely difficult to achieve convergence of the iterative method for the nonlinear equation. Moreover, for some values of the starting guess for the initial value, the solution of the IVP may not exist over the entire interval of integration in that the solution may become unbounded before reaching the right-hand endpoint of the BVP.

A potential remedy for the difficulties associated with simple shooting is provided by **multiple shooting**, in which the interval of integration $[a, b]$ is divided into subintervals and shooting is carried out on each subinterval separately.

Requiring continuity at the internal mesh points provides boundary conditions for the individual subproblems. Restricting the length of its interval of integration improves the conditioning of each IVP, but it also results in a larger system of nonlinear equations to solve. Specifically, the new system of ODEs is of size mn, where m is the number of subintervals and n is the size of the original system.

Multiple shooting requires starting guesses for the initial values and slopes at the mesh points, and it also requires some new choices, such as the number of subintervals to use. Although it is more robust than simple shooting, multiple shooting is hardly foolproof and must be used with considerable care.

CHAPTER 10

Partial Differential Equations (PDEs)

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import animation, cm
from scipy import integrate, linalg, sparse
```

10.1 Introduction

A **partial differential equation, or PDE**, is an equation involving partial derivatives of an unknown function with respect to more than one independent variable. PDEs are of fundamental importance in modeling all types of continuous phenomena in nature.

Many of the basic laws of science are expressed as PDEs, including:

- Maxwell's equations, which describe the behavior of an electromagnetic field by prescribing the relationships among the electric and magnetic field strengths, magnetic flux density, and electric charge and current densities.

$$\nabla \cdot D = \frac{\rho}{\epsilon_0}$$
$$\nabla \cdot B = 0$$
$$\nabla \times E = -\frac{\partial B}{\partial Bt}$$
$$\nabla \times B = \mu_0 J + \frac{\partial E}{c^2 \partial Bt}$$

- Navier-Stokes equations, which describe the behavior of a fluid by prescribing the relationships among its velocity, density, pressure, and viscosity.

$$\nabla \cdot u = 0$$
$$\rho \frac{\partial u}{\partial t} = -\nabla p + \mu \nabla^2 u + \rho F$$

- Schrödinger's equation of quantum mechanics, which describes the wave function of a particle by prescribing the relationships among its mass, potential energy, and total energy.

$$i\hbar \frac{\partial}{\partial t} \Psi = \left[\frac{-\hbar^2}{2m} \nabla^2 + V \right] \Psi$$

- Einstein's equations of general relativity, which describe a gravitational field by prescribing the relationship between the curvature of spacetime and the energy density of the matter it contains.
- Linear elasticity equations, which describe vibrations in an elastic solid with given material properties by prescribing the relationship between stress and strain.

We will confine our attention to PDEs that are simpler than those just listed, as the general theory of PDEs is far beyond the scope of this course.

We will consider some basic concepts and methods in relatively simple settings, but most of these are applicable more generally. Many of these ideas carry over from ODEs, such as the need to specify initial or boundary conditions, but the situation is typically more complex with PDEs, in part because a problem domain in two or more dimensions can be much more irregular, and the boundary conditions much more complicated, than in one dimension.

Many of the numerical solution techniques we saw in the ODE notebook also carry over to PDEs, but the computational cost increases substantially with the number of independent variables because the system of algebraic equations resulting from discretization becomes much larger.

10.1.1 Notation

For simplicity, we will deal only with single PDEs (as opposed to systems of several coupled PDEs) with only two independent variables:

- One space variable denoted by x and a time variable denoted by t (analogous to the initial value problems for ODEs)
- Two space variables denoted by x and y (analogous to the boundary value problems for ODEs).

We denote the unknown solution function by u, and we denote its partial derivatives with respect to the independent variables by appropriate subscripts:

- $u_t = \partial u/\partial t$
- $u_{xy} = \partial^2 u/\partial x \partial y$
- $\cdots$

We seek to determine a function u whose partial derivatives with respect to the independent variables satisfy the relationship prescribed by a given PDE on a given domain, and which also satisfies whatever initial or boundary conditions may have been imposed.

Such a solution function u can be visualized as a surface over the relevant two-dimensional domain in the (t, x) or (x, y) plane.

10.2 Classification and examples

As with ODEs, the order of a PDE is determined by the highest-order partial derivative appearing in the PDE.

Some of the most important PDEs in practical applications are of second order, including

- **Heat equation**

$$u_t = u_{xx}$$

- **Wave equation**

$$u_{tt} = u_{xx}$$

- **Laplace equation**

$$u_{xx} + u_{yy} = 0$$

It turns out that these three equations are general prototypes in that any second-order linear PDE of the form

$$au_{xx} + bu_{xy} + cu_{yy} + du_x + eu_y + fu + g = 0$$

can be transformed by a change of variables into one of these three canonical equations (plus terms of lower order), provided a, b, and c are not all zero.

The quantity $b^2 - 4ac$, called the discriminant, determines which of the canonical forms is obtained by such a transformation.

Therefore, second-order linear PDEs can be classified according to the value of the discriminant into three families whose names derive from the analogous conic sections:

- $b^2 - 4ac > 0$: **hyperbolic**, typified by the wave equation
- $b^2 - 4ac = 0$: **parabolic**, typified by the heat equation
- $b^2 - 4ac < 0$: **elliptic**, typified by the Laplace equation

> **Where do these names come from?**
>
> The equation
>
> $$au_{xx} + bu_{xy} + cu_{yy} + du_x + eu_y + fu + g = 0$$
>
> is reminiscent of the equation
>
> $$ax^2 + bxy + cy^2 + dx + ey$$
>
> which is the general form for *conic sections*, where the prototypical hyperbola, parabola and ellipse are given by
>
> $$x^2 - y^2 = 1$$
> $$y = x^2$$
> $$x^2 + y^2 = 1$$

Generally,

- *Hyperbolic* PDEs describe time-dependent, conservative physical processes, such as convection, that are not evolving toward a steady state.
- *Parabolic* PDEs describe time-dependent, dissipative physical processes, such as diffusion, that are evolving toward a steady state.
- *Elliptic* PDEs describe systems that have already reached a steady state, or equilibrium, and hence are time-independent.

Systems governed by *hyperbolic* PDEs are conservative in that the "energy" of the system, as measured by an appropriate norm of the solution, is conserved over time. Hyperbolic PDEs are analogous to a linear system of ODEs whose matrix has purely imaginary eigenvalues, yielding a purely oscillatory solution that neither grows nor decays with time.

Systems governed by *parabolic* PDEs, on the other hand, are dissipative in that the "energy" of the solution diminishes over time. Parabolic PDEs are analogous to a linear system of ODEs whose matrix has only eigenvalues with negative real parts, yielding an exponentially decaying solution.

Another important difference is that hyperbolic PDEs propagate information at a finite speed, whereas parabolic PDEs propagate information instantaneously.

These differences between the two types of time-dependent PDEs have important theoretical and practical implications.

For example, parabolic PDEs have a smoothing effect that over time damps out any lack of smoothness in the initial conditions, whereas hyperbolic PDEs propagate steep fronts or shocks undiminished, and discontinuities can develop in the solution even with smooth initial data.

Systems governed by hyperbolic PDEs are in principle reversible in time, whereas parabolic systems are not. The heat equation integrated backward in time is ill-posed, for example, which corresponds physically to the fact that one cannot determine details of the thermal history of a system from its current temperature distribution.

The challenges in solving parabolic or hyperbolic PDEs numerically are analogous to those in solving ODEs that are stiff because of eigenvalues with large negative real parts (parabolic) or large imaginary parts (hyperbolic).

10.3 Solving time-dependent problems

Numerical methods for time-dependent PDEs typically use discrete time-stepping procedures to generate an approximate solution step-by-step in time, analogous to the methods for ODE initial value problems. The corresponding spatial discretization can be accomplished in a number of ways

The 2 most common are:

- **Semidiscrete methods**

> One way to approximate the solution to a time-dependent PDE numerically is to discretize in space but leave the time variable continuous. This approach results in a system of ODEs, which can then be solved by the methods discussed in the ODE notebook.

- **Fully discrete methods**

> In a fully discrete method, all of the independent variables in the PDE are discretized, including time.
>
> In a fully discrete finite difference method, we introduce a grid of mesh points throughout the problem domain in space and time, we replace all the derivatives in the PDE by finite difference approximations, and we seek an approximate value for the solution at each of the mesh points. The resulting array of approximate solution values represents a discrete sample of points on the solution surface over the problem domain in the (t, x) plane. The accuracy of such an approximate solution depends on the step sizes in both space and time.
>
> Replacement of all partial derivatives by finite difference approximations results in a system of algebraic equations for the unknown solution values at the discrete set of mesh points.
>
> This system may be linear or nonlinear, depending on the underlying PDE. With an initial-value problem, the solution is obtained by beginning with the initial values along an appropriate boundary of the problem domain and then marching forward step by step in time, generating successive rows in the solution array.
>
> Such a time-stepping procedure may be **explicit** or **implicit**, depending on whether the formula for the solution values at the next time step involves only current and past information.

Practical examples of both discretization methods are shown below for specific examples of the Heat and Wave equation.

10.3.1 Heat equation

The heat equation in one space dimension has the form

$u_t = cu_{xx}, 0 \leq x \leq L, t \geq 0,$

with given initial condition $u(0, x) = f(x), 0 \leq x \leq L,$

and boundary conditions $u(t, 0) = \alpha, u(t, L) = \beta\ , t \geq 0.$

and c a positive constant.

This equation models, for example, the diffusion of heat in a bar of length L whose ends are maintained at temperatures specified by the boundary conditions and whose initial temperature distribution is given by the function $f(x)$. The constant c, which governs the rate of diffusion, depends on material properties of the bar, such as its thermal conductivity, specific heat, and density.

The solution u to this equation gives the subsequent temperature distribution as a function of both space and time

Example: solving the heat equation with a semidiscrete method

Consider the equation, describing the diffusion of heat in a 1 m long bar, with one end kept at 0 K, and the other at 1 K.

The initial temperature distribution is given by the superposition of a straight line and a Gaussian curve $f(x) = x + \exp - \left(\frac{(x-0.5)}{0.1}\right)^2$

$u_t = cu_{xx}$, $0 \leq x \leq 1$, $t \geq 0$,

with given initial condition $u(0, x) = f(x)$, $0 \leq x \leq L$,

and boundary conditions $u(t, 0) = 0$, $u(t, 1) = 1$, $t \geq 0$.

with c the thermal diffusivity of the material (e.g. 23 mm^2/s for iron).

If we introduce spatial mesh points $x_i = i\Delta x$, $i = 0, \ldots, n+1$, where $\Delta x = 1/(n+1)$, and replace the second derivative u_{xx} with the finite difference approximation

$$u_{xx}(t, x_i) \approx \frac{u(t, x_{i+1}) - 2u(t, x_i) + u(t, x_{i-1})}{(\Delta x)^2} \quad , \quad i = 1, \ldots, n$$

but leave the time variable continuous, then we obtain a system of ODEs

$$y_i'(t) = \frac{c}{(\Delta x)^2}(y_{i+1}(t) - 2y_i(t) + y_{i-1}(t)) \quad , \quad i = 1, \ldots, n$$

where $y_i(t) \approx u(t, x_i)$. From the boundary conditions we know that, $y_0(t) = 0$ and $y_{n+1}(t) = 1$, and from the initial conditions, that $y_i(0) = f(x_i)$, $i = 1, \ldots, n$

We can therefore use an ODE method to solve the initial value problem for this system.

This approach is sometimes called **the method of lines**. If we think of the solution $u(t, x)$ as a surface over the (t, x) plane, this method computes cross sections of that surface along a series of lines, each of which passes through one of the discrete spatial mesh points and runs parallel to the time axis.

The foregoing semidiscrete system of ODEs can be written in matrix form as

$$\mathbf{y}' = \frac{c}{(\Delta x)^2}\begin{bmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & \cdots & 0 \\ 0 & 1 & -2 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 & -2 \end{bmatrix}\mathbf{y} = \mathbf{A}\mathbf{y}$$

The Jacobian matrix $\mathbf{A}$ of this system has eigenvalues between $-4c/(\Delta x)^2$ and 0, which makes the ODE very stiff as the spatial mesh size Δx becomes small. This stiffness, which is typical of ODEs derived from PDEs in this manner, must be taken into account in choosing an appropriate ODE method for solving the semidiscrete system.

```
def demo_heat_equation():
    """Define and solve heat equation example with a semidiscrete method."""
    # define spatial discretization and thermal diffusion constant
    N = 100
    c = 23e-6

    # define A
    A = (
        sparse.diags(np.ones(N - 1), -1).toarray()
        + sparse.diags(np.ones(N) * -2, 0).toarray()
        + sparse.diags(np.ones(N - 1), 1).toarray()
```

(continues on next page)

(continued from previous page)

```
    )
    print(A)
    A = A * c * (N + 1) ** 2

    # boundary conditions: do not change edge points
    A[0, :] = 0
    A[N - 1, :] = 0

    # y'=Ay (nice and simple once you have A)
    def func(t, y):
        return A @ y

    # Discretize our rod
    x = np.linspace(0, 1, N)
    # Define initial temperature profile and plot it
    y0 = np.exp(-1 * ((x - 0.5) / 0.1) ** 2) + x

    # Use an ODE solver to solve this problem
    sol = integrate.solve_ivp(func, [0, 2500], y0, t_eval=np.arange(0, 2501,␣
↪10))

    # --- Everything below makes an animation of the solution ---

    # First set up the figure, the axis, and the plot element we want to␣
↪animate
    fig = plt.figure(num="heat", clear=True)
    ax = plt.axes(xlim=(0, 1), ylim=(0, 1.5))
    ax.plot(x, y0, ".", label="initial")
    (line,) = ax.plot([], [], lw=2, label="time-dependent")
    ax.legend()

    # initialization function: plot the background of each frame
    def init():
        line.set_data([], [])
        return (line,)

    # animation function.
    def animate(i):
        x = np.linspace(0, 1, N)
        y = sol.y[:, i]
        line.set_data(x, y)
        return (line,)

    # call the animator.  blit=True means only re-draw the parts that have␣
↪changed.
    return animation.FuncAnimation(
        fig,
        animate,
        init_func=init,
        frames=250,
        interval=20,
        blit=True,
        repeat=False,
    )
```

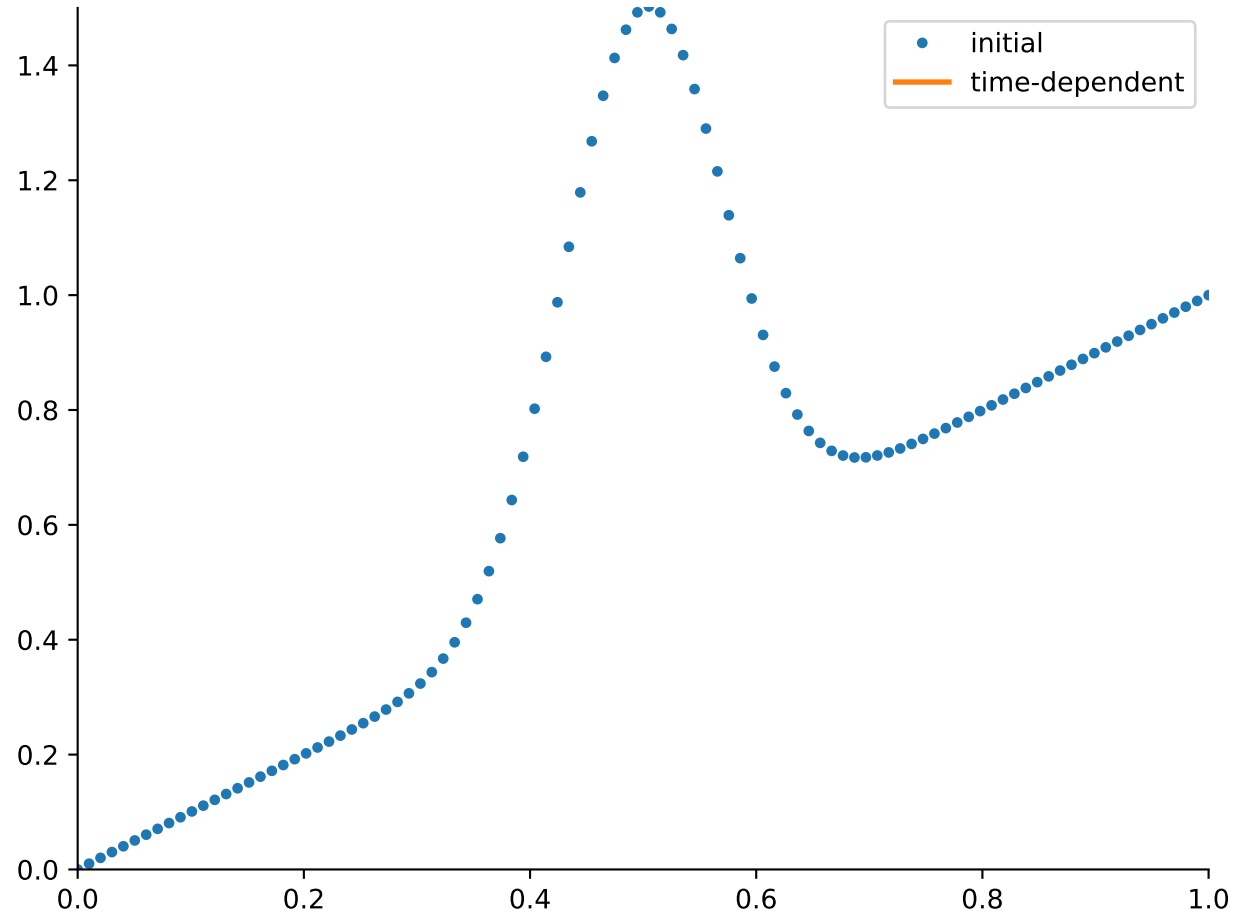
(continues on next page)

(continued from previous page)

```
plt.close("heat")
demo_heat_equation()
```

```
[[-2.  1.  0. ...  0.  0.  0.]
 [ 1. -2.  1. ...  0.  0.  0.]
 [ 0.  1. -2. ...  0.  0.  0.]
 ...
 [ 0.  0.  0. ... -2.  1.  0.]
 [ 0.  0.  0. ...  1. -2.  1.]
 [ 0.  0.  0. ...  0.  1. -2.]]
```

```
<matplotlib.animation.FuncAnimation at 0x7f76ee5d6f90>
```



10.3.2 Wave equation

The wave equation in one spatial dimension has the form

$u_{tt} = cu_{xx}$, $0 \leq x \leq L, t \geq 0$,

with given initial conditions $u(0, x) = f(x)$, $u_t(0, x) = g(x)$, $0 \leq x \leq L$,

and boundary conditions $u(t, 0) = \alpha$, $u(t, L) = \beta$, $t \geq 0$,

and c a positive constant.

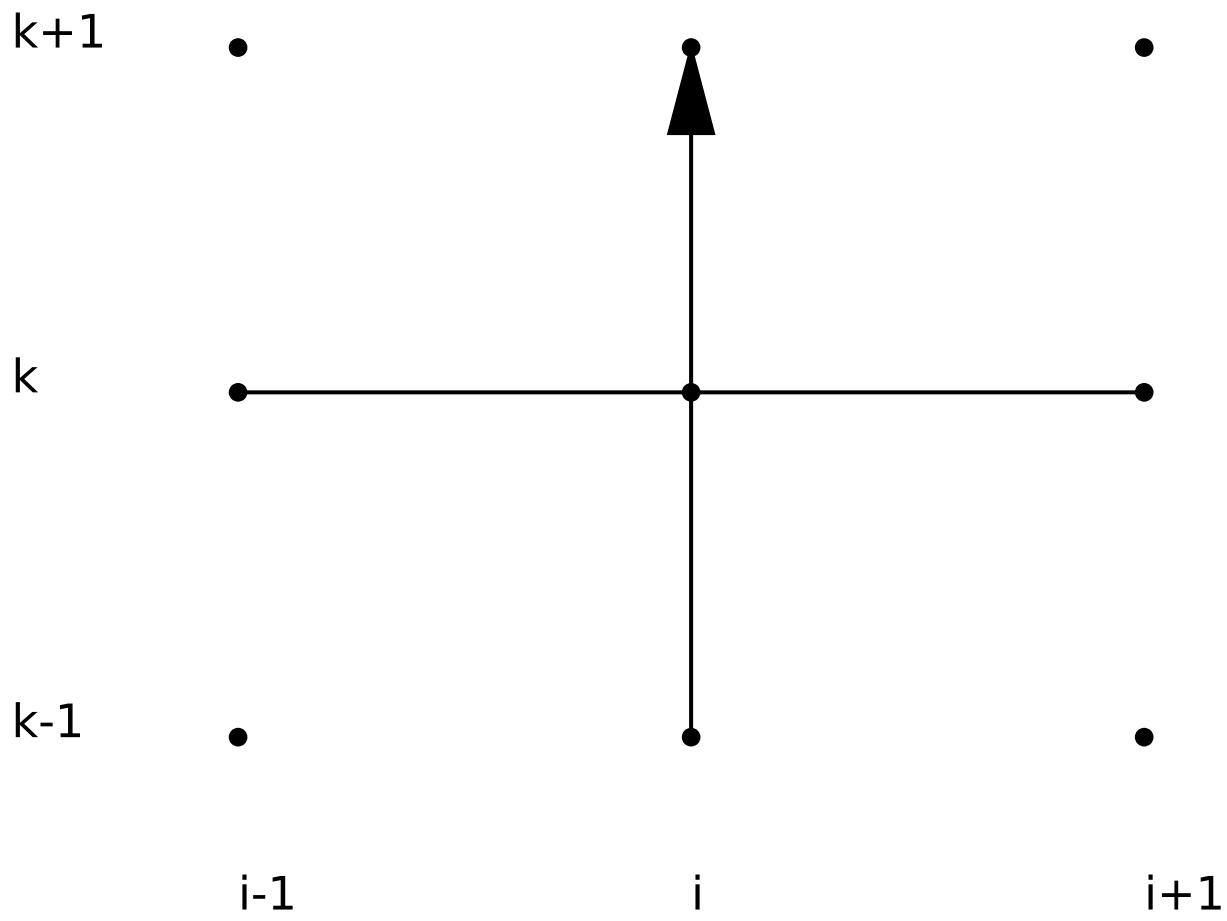
This equation models, for example, vibrations of a violin string of length L whose initial profile and velocity are given by the functions $f(x)$ and $g(x)$, respectively, and whose ends are anchored as prescribed by the boundary conditions.

Because it is second-order in time, this equation requires initial conditions for both the solution function and its first derivative with respect to time. The solution consists of waves propagating both to the left and the right with speed $\sqrt{c}$.

> **Example: solving the wave equation with a fully discrete method**
>
> We define spatial mesh points $x_i = i\Delta x$, $i = 0, 1, \ldots, n+1$, where $\Delta x = L/(n+1)$, and temporal mesh points $t_k = k\Delta t$, $k = 0, 1, \ldots$, where Δt is chosen appropriately.
>
> We denote the approximate solution at mesh point (t_k, x_i) by u_i^k , where we have used both a subscript and a superscript (the k is not an exponent) to distinguish clearly between increments in space and time, respectively.
>
> Using centered difference approximations for both u_{tt} and u_{xx} yields a system of algebraic equations
>
> $$\frac{u_i^{k+1} - 2u_i^k + u_i^{k-1}}{(\Delta t)^2} = c\frac{u_{i+1}^k - 2u_i^k + u_{i-1}^k}{(\Delta x)^2} \quad , \quad i = 1, \ldots, n$$
>
> which can be rearranged to give the explicit recurrence
>
> $$u_i^{k+1} = 2u_i^k - u_i^{k-1} + c\left(\frac{\Delta t}{\Delta x}\right)^2 (u_{i+1}^k + 2u_i^k + u_{i-1}^k)$$

The pattern of mesh points involved in computing u_i^{k+1} is illustrated in the figure below, where lines connect the relevant mesh points and an arrow indicates the mesh point at which the approximate solution is being computed. Such a pattern is called the **stencil** of a given finite difference scheme.

```
def plot_stencil_1():
    """Plot the stencil for the explicit method for the wave equation."""
    plt.close("stencil1")
    fig, ax = plt.subplots(num="stencil1")
    ax.plot(np.arange(3), np.zeros(3), "o", c="k")
    ax.plot(np.arange(3), np.ones(3), "o", c="k")
    ax.plot(np.arange(3), 2 * np.ones(3), "o", c="k")
    ax.text(-0.5, 0, "k-1", fontsize="xx-large")
    ax.text(-0.5, 1, "k", fontsize="xx-large")
    ax.text(-0.5, 2, "k+1", fontsize="xx-large")

    ax.text(0, -0.5, "i-1", fontsize="xx-large")
    ax.text(1, -0.5, "i", fontsize="xx-large")
    ax.text(2, -0.5, "i+1", fontsize="xx-large")

    ax.arrow(1, 0, 0, 1.75, fc="k", ec="k", head_width=0.10, head_length=0.25)

    ax.axis("off")

    ax.plot([0, 2], [1, 1], c="k")
    ax.plot([1, 1], [0, 2], c="k")


plot_stencil_1()
```



This scheme is second-order accurate in both space and time, but it requires data at two successive time levels, which means that additional storage is required, and it also means that we need both u_i^0 and u_i^1 initially to get started. These values can be obtained from the initial conditions $u_i^0 = f(x_i)$, $u_i^1 = u_i^0 + \Delta t g(x_i)$, $i = 1, \ldots, n$, where in the latter we have used a forward difference approximation to the initial condition $u_t(0, x) = g(x)$, $0 \leq x \leq 1$.

> To make this more specific, let's consider the wave equation
>
> $$\frac{\partial^2 f(x,t)}{\partial x^2} = \frac{\rho}{T} \frac{\partial^2 f(x,t)}{\partial t^2}$$
>
> or, in our notation,
>
> $$u_{xx} = \frac{\rho}{T} u_{tt}$$
>
> describing the waves on a string with a density ρ (in kg/m) under tension T (in N).
>
> with the boundary conditions that the string is attached at $x = 0$ and $x = 1$.

```
def check_stability_criterion(c, delta_x, delta_t):
    """Check if the finite difference method is stable.

    This function is used by `demo_wave_equation()` below.

    Parameters
```

(continues on next page)

(continued from previous page)

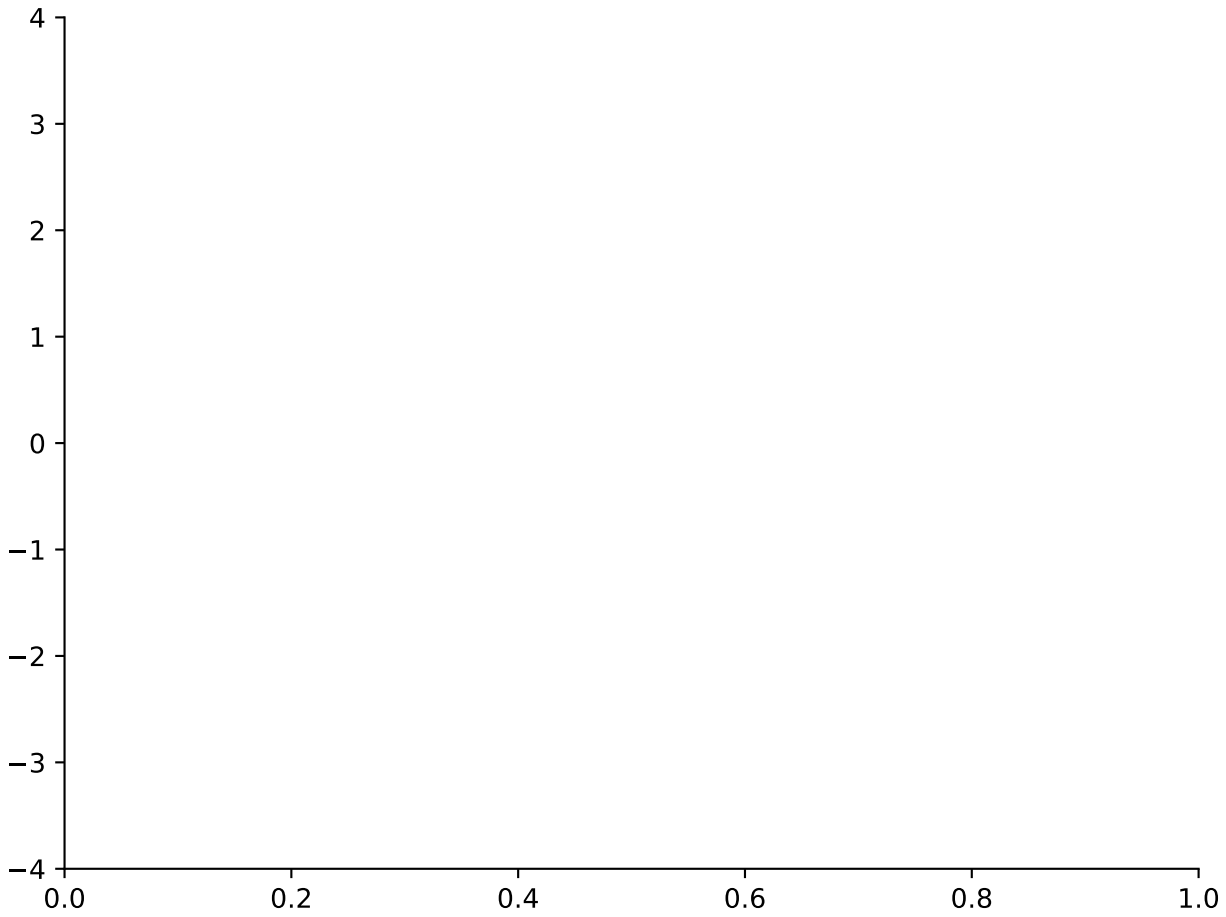
```
    ----------
    c : float
        The speed of the wave
    delta_x : float
        The distance between each spatial point
    delta_t : float
        The time step

    Returns
    -------
    bool
        True if the method is stable, False otherwise
    """
    stability_criterion = c * (delta_t / delta_x) ** 1
    if stability_criterion > 1:
        print("Warning: The method is unstable. Consider using smaller time␣
↪steps.")
    return stability_criterion <= 1


def demo_wave_equation():
    """Define and solve a wave equation example with a finite-difference␣
↪method."""

    # define spatial discretization
    N = 100
    c = 1

    # define our 1D-grid
    x = np.linspace(0.0, 1.0, N)
    delta_x = x[1] - x[0]

    # we will model a standing wave where the displacement
    # is given analytically by  y=sin(2 pi x)cos(2 pi ct)
    # at time t=0 we get
    y = np.sin(x * 2 * np.pi)
    # which has dy/dt= -2*pi*sin(2 p i x) as velocity
    g = -2.0 * np.pi * np.sin(x * 2 * np.pi)

    # time "grid"
    t_range = np.linspace(0, 2, 200)
    delta_t = t_range[1] - t_range[0]

    # define A
    A = (
        sparse.diags(np.ones(N - 1), -1).toarray()
        + sparse.diags(np.ones(N) * -2, 0).toarray()
        + sparse.diags(np.ones(N - 1), 1).toarray()
    )
    A = A * c * (delta_t / delta_x) ** 2

    check_stability_criterion(c, delta_x, delta_t)

    # boundary conditions: do not change edge points
```

(continues on next page)

(continued from previous page)

```
    A[0, :] = 0
    A[N - 1, :] = 0

    # first step using y'
    yprev = y
    y = y + delta_t * g

    # initialize an empty array in which we'll store the solutions
    solutions = np.zeros((len(t_range) - 1, N))

    # Do the actual work.
    # Note the helper variable 'ybetween' because
    # we need information of a previous time point.
    for t in range(len(t_range) - 1):
        ybetween = y
        y = 2 * y - yprev + A @ y
        solutions[t, :] = y
        yprev = ybetween

    # --- Everything below makes an animation of the solution ---

    # First set up the figure, the axis, and the plot element we want to␣
↪animate
    fig = plt.figure(num="wave", clear=True)
    ax = plt.axes(xlim=(0, 1), ylim=(-4, 4))
    (line,) = ax.plot([], [], lw=2)

    # initialization function: plot the background of each frame
    def init():
        line.set_data([], [])
        return (line,)

    # animation function.
    def animate(i):
        x = np.linspace(0, 1, N)
        y = solutions[i, :]
        line.set_data(x, y)
        return (line,)

    # call the animator.  blit=True means only re-draw the parts that have␣
↪changed.
    return animation.FuncAnimation(
        fig,
        animate,
        init_func=init,
        frames=(len(t_range) - 1),
        interval=20,
        blit=True,
        repeat=False,
    )


plt.close("wave")
demo_wave_equation()
```

```
<matplotlib.animation.FuncAnimation at 0x7f76eb8739d0>
```



In principle, there is no real distinction between discrete and semidiscrete methods for time-dependent PDEs, since the time variable is ultimately discretized in either case. There is an important practical distinction, however, in that with a semidiscrete method we entrust to a sophisticated, adaptive ODE software package the responsibility for choosing time step sizes that will maintain stability and attain the desired accuracy, whereas with a fully discrete method, the user must explicitly choose time step sizes to achieve these same goals.

10.3.3 Implicit methods: heat equation revisited

As with ODEs, a larger stability region that permits larger time steps can be obtained by using implicit methods. For the heat equation, for example, applying the backward Euler method to the semidiscrete system shown in the Heat equation example yields the implicit finite difference scheme

$$u_i^{k+1} = u_i^k + c\frac{\Delta t}{(\Delta x)^2}(u_{i+1}^{k+1} - 2u_i^{k+1} + u_{i-1}^{k+1}) \quad , \quad i = 1, \ldots, n$$

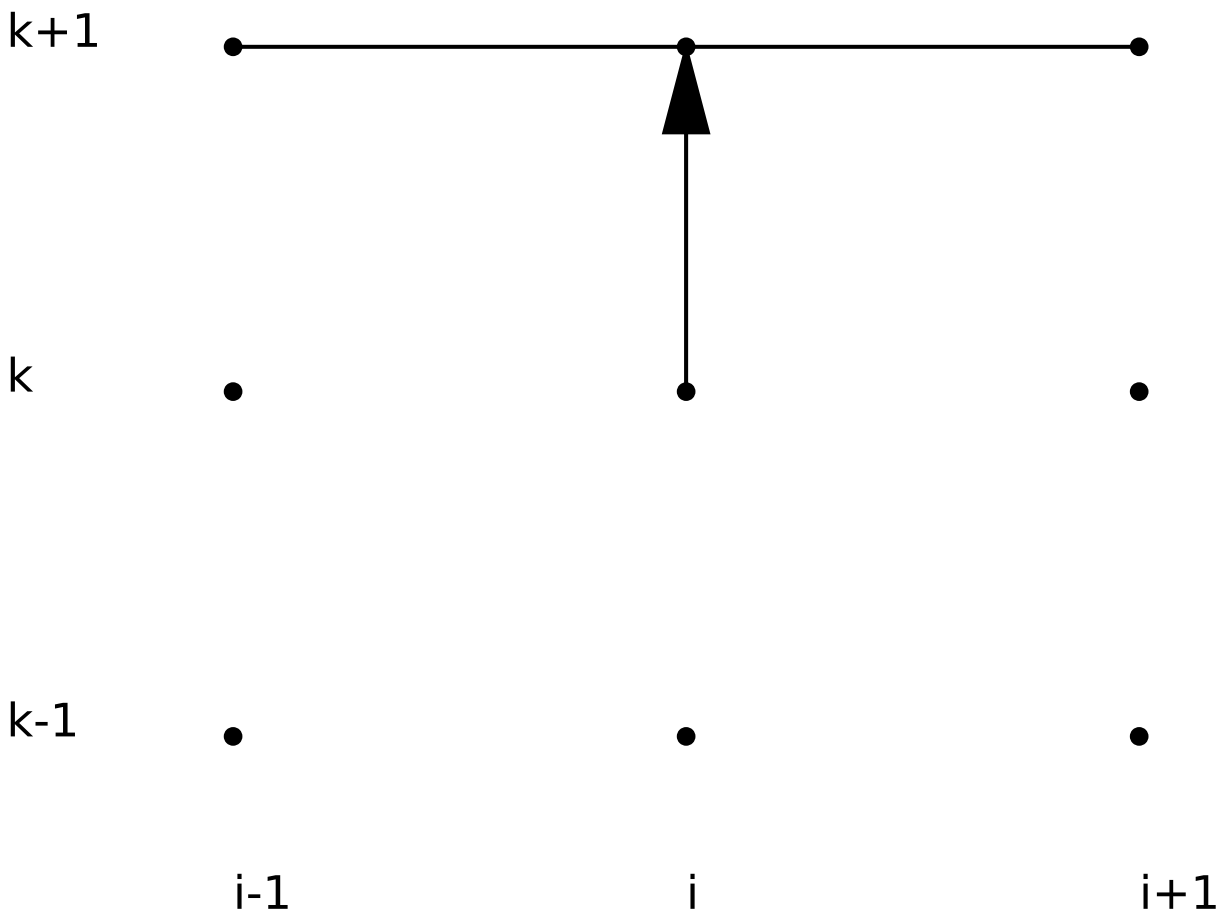
whose stencil is shown here:

```
def plot_stencil_2():
    """Plot the stencil for the implicit method for the heat equation."""
    plt.close("stencil2")
    fig, ax = plt.subplots(num="stencil2")
    ax.plot(np.arange(3), np.zeros(3), "o", c="k")
    ax.plot(np.arange(3), np.ones(3), "o", c="k")
    ax.plot(np.arange(3), 2 * np.ones(3), "o", c="k")
```

(continues on next page)

(continued from previous page)

```
    ax.text(-0.5, 0, "k-1", fontsize="xx-large")
    ax.text(-0.5, 1, "k", fontsize="xx-large")
    ax.text(-0.5, 2, "k+1", fontsize="xx-large")

    ax.text(0, -0.5, "i-1", fontsize="xx-large")
    ax.text(1, -0.5, "i", fontsize="xx-large")
    ax.text(2, -0.5, "i+1", fontsize="xx-large")

    ax.arrow(1, 1, 0, 0.75, fc="k", ec="k", head_width=0.10, head_length=0.25)

    ax.axis("off")

    ax.plot([0, 2], [2, 2], c="k")
    ax.plot([1, 1], [1, 2], c="k")


plot_stencil_2()
```



In the demo below we again solve the heat equation described in section 3.1, but now using the implicit method whose stencil is shown above.

```
def demo_heat_equation_implicit():
    """Define and solve a heat equation example with an implicit method."""
    # define spatial and temporal discretizaiton
```

(continues on next page)

(continued from previous page)

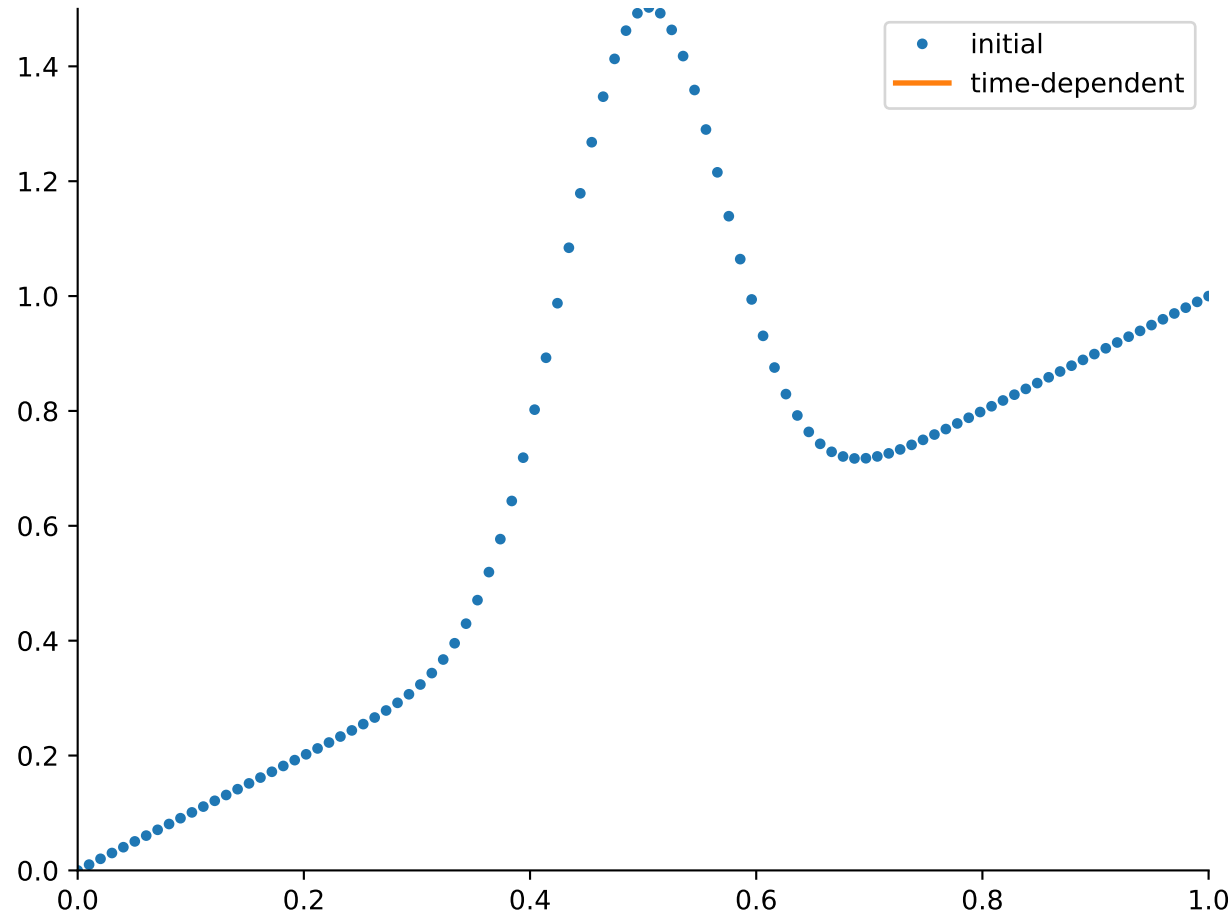
```
    Nx = 100
    Nt = 1000
    c = 23e-6

    # define A
    A = (
        sparse.diags(np.ones(Nx - 1), -1).toarray()
        + sparse.diags(np.ones(Nx) * -2, 0).toarray()
        + sparse.diags(np.ones(Nx - 1), 1).toarray()
    )
    A = A * c * (Nx + 1) ** 2

    # boundary conditions: do not change edge points
    A[0, :] = 0
    A[Nx - 1, :] = 0

    # y'=Ay (nice and simple once you have A)
    def func(t, y):
        return A @ y

    # Discretize our rod
    x = np.linspace(0, 1, Nx)
    # Define initial temperature profile and plot it
    y0 = np.exp(-1 * ((x - 0.5) / 0.1) ** 2) + x

    # initialize an empty array in which we'll store the solutions
    sol = np.zeros((Nt, Nx))

    y = y0
    # do the actual work.
    for t in np.arange(Nt):
        # the equation we're trying to solve is written implicitly as
        # ynext = y+A@ynext
        # this can be written in the form Ax=b as [A-1]ynext=-y
        # which can then be solved using linalg.solve:

        ynext = linalg.solve(A - sparse.diags(np.ones(Nx), 0).toarray(), -1 *␣
↪y)

        sol[t, :] = ynext
        y = ynext

    # ---Everything below makes an animation of the solution ---

    # First set up the figure, the axis, and the plot element we want to␣
↪animate
    fig = plt.figure(num="heat_implicit", clear=True)
    ax = plt.axes(xlim=(0, 1), ylim=(0, 1.5))
    ax.plot(x, y0, ".", label="initial")
    (line,) = ax.plot([], [], lw=2, label="time-dependent")
    ax.legend()

    # initialization function: plot the background of each frame
    def init():
```

(continues on next page)

(continued from previous page)

```
        line.set_data([], [])
        return (line,)

    # animation function.
    def animate(i):
        x = np.linspace(0, 1, Nx)
        y = sol[i, :]
        line.set_data(x, y)
        return (line,)

    # call the animator.  blit=True means only re-draw the parts that have␣
↪changed.
    return animation.FuncAnimation(
        fig, animate, init_func=init, frames=Nt, interval=2, blit=True,␣
↪repeat=False
    )


plt.close("heat_implicit")
demo_heat_equation_implicit()
```

```
<matplotlib.animation.FuncAnimation at 0x7f76eb709810>
```



The scheme used above inherits the unconditional stability of the backward Euler method, which means that there is no

stability restriction on the relative sizes of Δt and Δx.

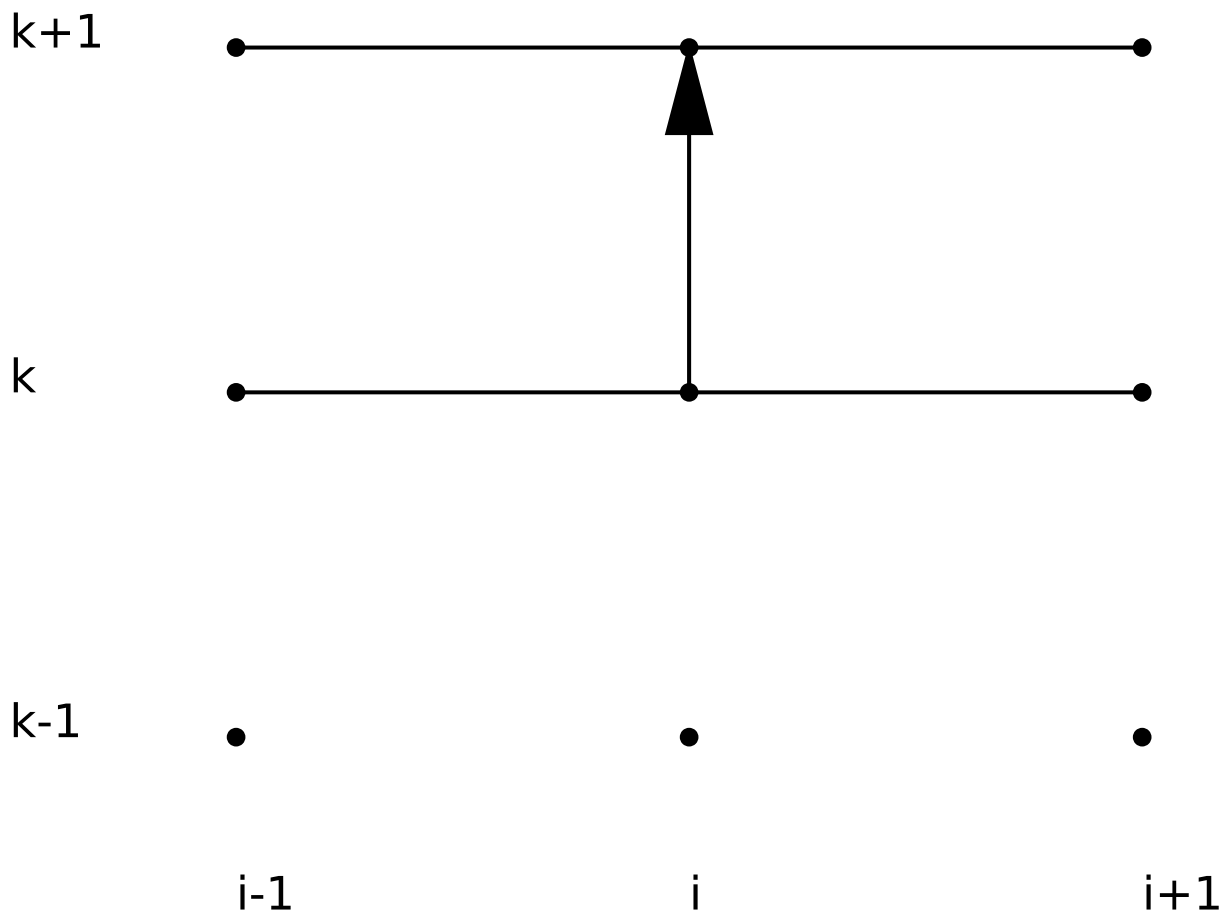
Accuracy is still a consideration, however, and the fact that this particular method is only first-order accurate in time still strongly limits the time step.

If instead we apply the trapezoid method we obtain the implicit finite difference scheme

$$u_i^{k+1} = u_i^k + c\frac{\Delta t}{2(\Delta x)^2}(u_{i+1}^{k+1} - 2u_i^{k+1} + u_{i-1}^{k+1} + u_{i+1}^k - 2u_i^k + u_{i-1}^k) \quad , \quad i = 1, \ldots, n$$

whose stencil is shown below.

```
def plot_stencil_3():
    """Plot the stencil for the Crank-Nicolson method for the heat equation."""
    plt.close("stencil3")
    fig, ax = plt.subplots(num="stencil3")
    ax.plot(np.arange(3), np.zeros(3), "o", c="k")
    ax.plot(np.arange(3), np.ones(3), "o", c="k")
    ax.plot(np.arange(3), 2 * np.ones(3), "o", c="k")
    ax.text(-0.5, 0, "k-1", fontsize="xx-large")
    ax.text(-0.5, 1, "k", fontsize="xx-large")
    ax.text(-0.5, 2, "k+1", fontsize="xx-large")

    ax.text(0, -0.5, "i-1", fontsize="xx-large")
    ax.text(1, -0.5, "i", fontsize="xx-large")
    ax.text(2, -0.5, "i+1", fontsize="xx-large")

    ax.arrow(1, 1, 0, 0.75, fc="k", ec="k", head_width=0.10, head_length=0.25)

    ax.axis("off")

    ax.plot([0, 2], [2, 2], c="k")
    ax.plot([0, 2], [1, 1], c="k")
    ax.plot([1, 1], [1, 2], c="k")


plot_stencil_3()
```



This scheme, called the **Crank-Nicolson method**, is unconditionally stable and is second-order accurate in time as well as in space.

The greater stability of implicit finite difference methods enables them to take much larger time steps than are permissible with explicit methods, but they require more work per step because we must solve a system of equations at each step to determine the approximate solution values.

10.4 Solving time-independent problems

Just as *time-dependent* parabolic and hyperbolic PDEs are analogous to *initial value problems* for ODEs, time-*independent* elliptic PDEs are analogous to *boundary-value problems* for ODEs, and most of the solution methods for ODE BVPs carry over to elliptic PDEs as well.

For an elliptic boundary value problem, the solution at every point in the problem domain depends on all of the boundary data (in contrast to the limited domain of dependence for time-dependent problems), and consequently an approximate solution must be computed everywhere simultaneously, rather than being generated step by step using a recurrence, as in the previous examples.

Consequently, discretization of an elliptic boundary value problem results in a single system of algebraic equations to be solved for some finite-dimensional approximation to the solution.

10.4.1 Laplace equation

The Laplace equation is a special case of the **Poisson equation**, which in two space dimensions has the form

$$u_{xx} + u_{yy} = f(x, y)$$

where f is a given function defined on a domain whose boundary is typically a closed curve in $\mathbb{R}^2$, such as a square or circle.

If $f \equiv 0$, then we have the **Laplace equation**.

There are numerous possibilities for the boundary conditions that must be specified on the boundary of the domain or portions thereof:

- **Dirichlet boundary conditions**, sometimes called essential boundary conditions, in which the solution u is specified.
- **Neumann boundary conditions**, sometimes called natural boundary conditions, in which one of the derivatives u_x or u_y is specified.
- **Robin boundary conditions**, or **mixed boundary conditions**, in which a combination of solution values and derivative values is specified.

The Laplace equation models, for example, the electrostatic potential within a charge-free region given the potential on the boundary of the region.

The Poisson equation models the electrostatic potential when there is also a known charge density within the region, represented by the function f.

For this reason, the Laplace equation or the Poisson equation is also sometimes called the **potential equation**.

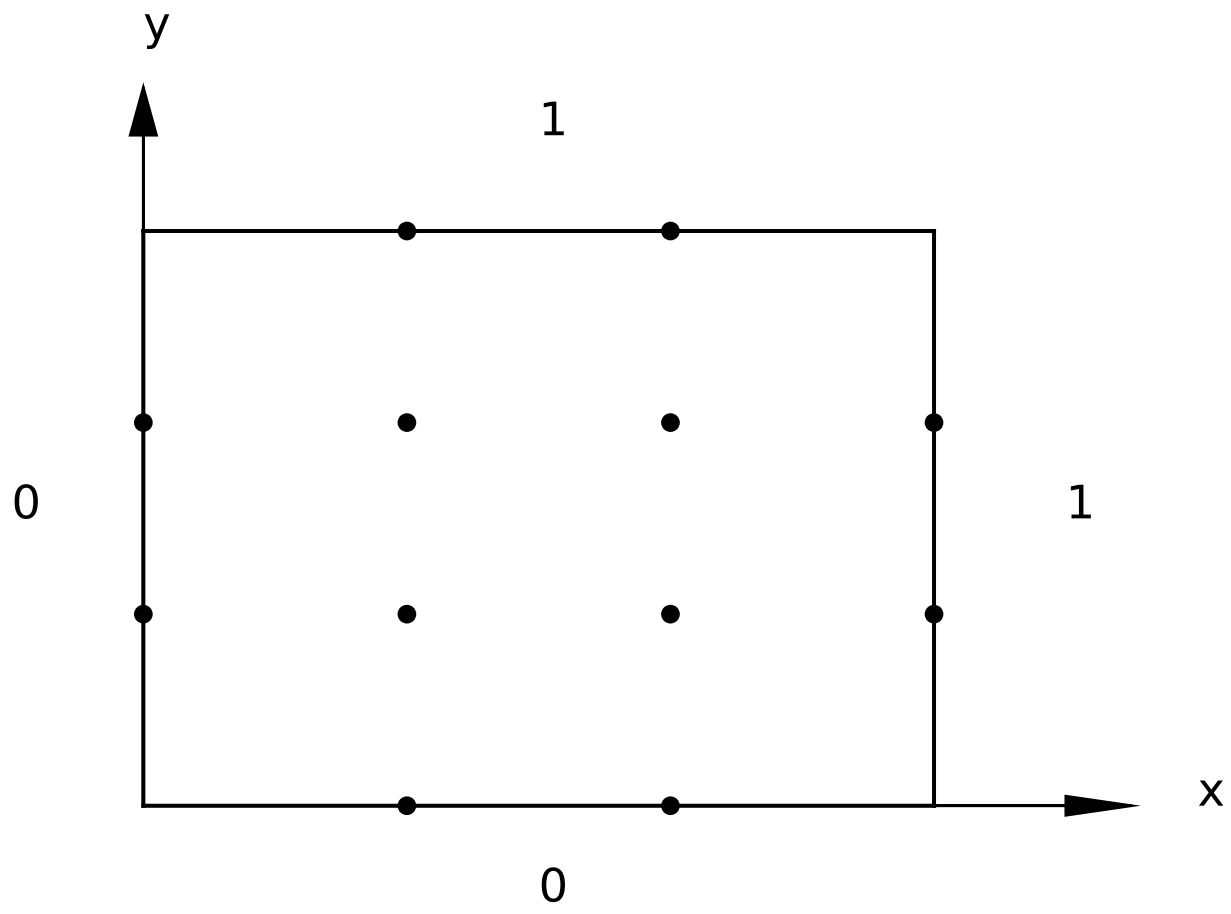
> **Example: solving the Laplace equation with a finite difference method**
>
> Finite difference methods for elliptic boundary value problems proceed as we have seen before: we define a discrete mesh of points within the problem domain and replace the derivatives in the PDE by finite difference approximations, but then we seek a numerical solution at all of the mesh points simultaneously by solving a single system of algebraic equations.
>
> Consider the Laplace equation on the unit square
>
> $$u_{xx} + u_{yy} = 0, \quad 0 \le x \le 1, \quad 0 \le y \le 1,$$
>
> We define a discrete mesh in the domain, including boundaries, as shown on the following figure, which also includes the boundary conditions.

```
def plot_laplace_mesh():
    """Plot the boundary conditions and mesh of the laplace equation example."
 ↪""
    plt.close("laplace_mesh")
    fig, ax = plt.subplots(num="laplace_mesh")
    ax.plot(np.arange(1, 3), np.zeros(2), "o", c="k")
    ax.plot(np.arange(4), np.ones(4), "o", c="k")
    ax.plot(np.arange(4), 2 * np.ones(4), "o", c="k")
    ax.plot(np.arange(1, 3), 3 * np.ones(2), "o", c="k")

    ax.text(-0.5, 1.5, "0", fontsize="xx-large")
    ax.text(1.5, -0.5, "0", fontsize="xx-large")
    ax.text(1.5, 3.5, "1", fontsize="xx-large")
    ax.text(3.5, 1.5, "1", fontsize="xx-large")

    ax.arrow(0, 0, 0, 3.5, fc="k", ec="k", head_width=0.10, head_length=0.25)
    ax.arrow(0, 0, 3.5, 0, fc="k", ec="k", head_width=0.10, head_length=0.25)

    ax.axis("off")
```

(continues on next page)

(continued from previous page)

```
    ax.plot([0, 3], [0, 0], c="k")
    ax.plot([0, 0], [0, 3], c="k")
    ax.plot([3, 3], [0, 3], c="k")
    ax.plot([0, 3], [3, 3], c="k")

    ax.text(0, 4, "y", fontsize="xx-large")
    ax.text(4, 0, "x", fontsize="xx-large")


plot_laplace_mesh()
```



The interior grid points where we will compute the approximate solution are given by

$$(x_i, y_j) = (ih, jh), \quad i, j = 1, \ldots, n$$

where in our example $n = 2$ and $h = 1/(n + 1) = 1/3$.

Next we replace the second derivatives in the equation with the standard second-order centered difference approximation at each interior mesh point to obtain the finite difference equations

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2} = 0 \quad , \quad i, j = 0, \ldots, n$$

where $u_{i,j}$ is an approximation to the true solution $u(x_i, y_j)$ and represents one of the given boundary values if i or j is 0 or $n + 1$.

Simplifying and writing out the resulting four equations explicitly, we obtain

$$\begin{aligned}
4u_{1,1} - u_{0,1} - u_{2,1} - u_{1,0} - u_{1,2} &= 0 \\
4u_{2,1} - u_{1,1} - u_{3,1} - u_{2,0} - u_{2,2} &= 0 \\
4u_{1,2} - u_{0,2} - u_{2,2} - u_{1,1} - u_{1,3} &= 0 \\
4u_{2,2} - u_{1,2} - u_{3,1} - u_{2,1} - u_{2,3} &= 0
\end{aligned}$$

Writing these four equations in matrix form, we obtain

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} 4 & -1 & -1 & 0 \\ -1 & 4 & 0 & -1 \\ -1 & 0 & 4 & -1 \\ 0 & -1 & -1 & 4 \end{bmatrix} \begin{bmatrix} u_{1,1} \\ u_{2,1} \\ u_{1,2} \\ u_{2,2} \end{bmatrix} = \begin{bmatrix} u_{0,1} + u_{1,0} \\ u_{3,1} + u_{2,0} \\ u_{0,2} + u_{1,3} \\ u_{3,2} + u_{2,3} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \mathbf{b}$$

This symmetric positive definite system of linear equations can be solved either by Cholesky factorization or by an iterative method, yielding the solution

$$\mathbf{x} = \begin{bmatrix} u_{1,1} \\ u_{2,1} \\ u_{1,2} \\ u_{2,2} \end{bmatrix} = \begin{bmatrix} 0.125 \\ 0.125 \\ 0.375 \\ 0.375 \end{bmatrix}$$

Note the symmetry in the solution, which reflects the symmetry in the problem, which we could have taken advantage of and solved a problem only half as large.

In a practical problem, the mesh size h would need to be much smaller to achieve acceptable accuracy in the approximate solution of the PDE, and the resulting linear system would be much larger than in the preceding example.

The matrix would be very sparse, however, since each equation would still involve at most only five of the variables, thereby saving substantially on work and storage.

In the next example you can change the parameter N to define smaller mesh sizes. The sparsity of the matrix A is visualized in the output, together with a 3D plot of the solution to the Laplace equation, using the same boundary conditions as in the minimal example above.

```
def demo_laplace_equation():
    """Define and solve a laplace equation example with a finite-difference␣
 ↪method.

    This generalizes the small problem we worked out analytically above.
    This code is partially copied from the poisson.py demo taught
    by Chris Rycroft in the Harvard Applied Math 205 course.
    """

    # define spatial discretizaiton
    N = 10
    # h = 1.0 / (N + 1)

    # define A
    A = np.zeros((N * N, N * N))

    for i in range(N):
        for j in range(N):
            ij = i + N * j
```

(continues on next page)

(continued from previous page)

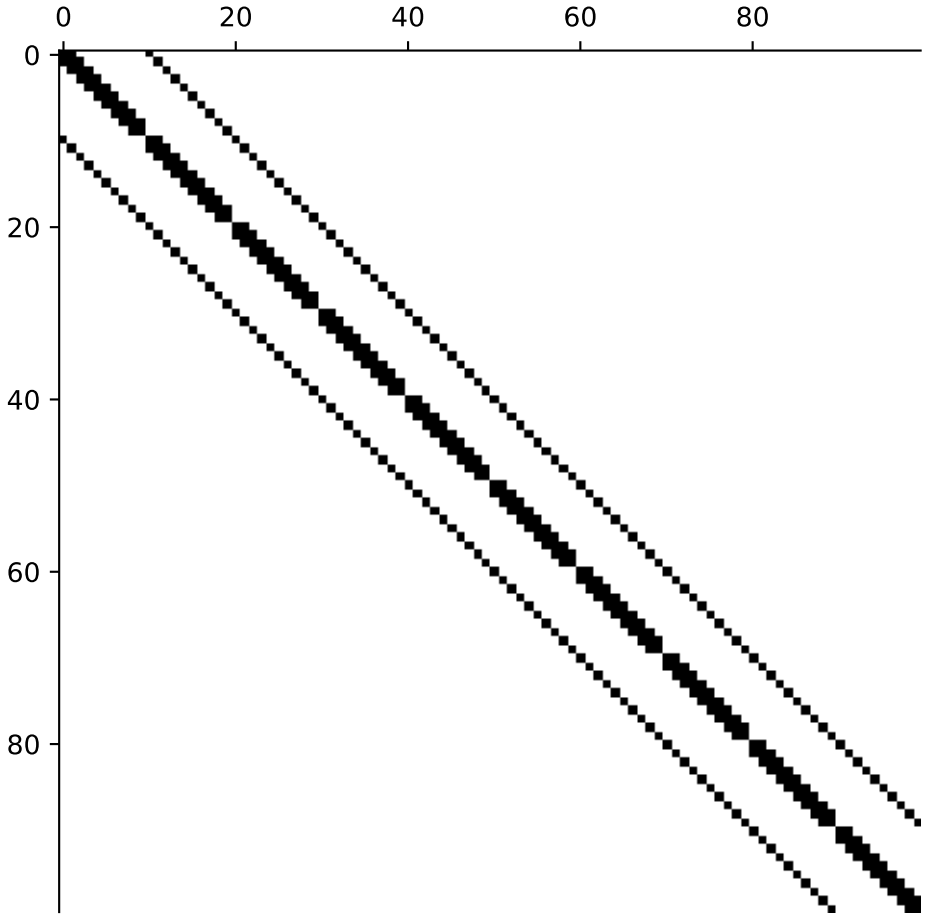
```
            A[ij, ij] = -4
            if i > 0.0:
                A[ij, ij - 1] = 1
            if i < N - 1:
                A[ij, ij + 1] = 1
            if j > 0.0:
                A[ij, ij - N] = 1
            if j < N - 1:
                A[ij, ij + N] = 1

    A *= -1

    # Display the sparsity structure of A
    print(A)
    plt.close("laplace_sparse")
    fig, ax = plt.subplots(num="laplace_sparse")
    ax.spy(A)
    ax.spines["top"].set_visible(True)
    ax.spines["bottom"].set_visible(False)
    ax.tick_params(axis="x", which="both", bottom=False)

    # define b, with boundary conditions as indictated in the book
    b = np.zeros(N * N)

    for i in range(N):
        for j in range(N):
            ij = i + N * j

            if i == 0.0:
                b[ij] = 0
            if i == N - 1:
                b[ij] = 1
            if j == 0.0:
                b[ij] = 0
            if j == N - 1:
                b[ij] = 1

    print(b)
    # solve the linear system
    u = np.linalg.solve(A, b)

    # --- Everything below makes an figure of the solution --

    uu = np.zeros((N + 2, N + 2))  # TODO add boundary conditions here
    for i in range(N):
        uu[i + 1, 1 : N + 1] = u[i * N : (i + 1) * N]
        uu[N + 1, :] = 1

    xa = np.linspace(0, 1, N + 2)
    mgx, mgy = np.meshgrid(xa, xa)
    plt.close("laplace")
    fig = plt.figure(num="laplace", clear=True)
    ax = fig.add_subplot(projection="3d")
```

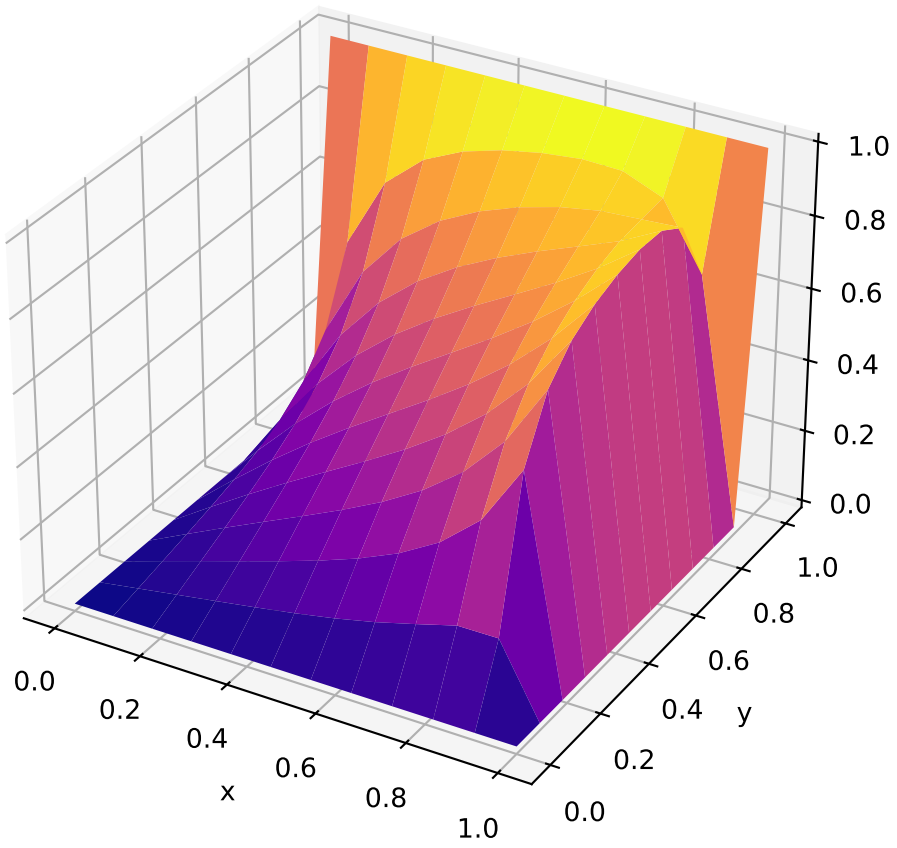
(continues on next page)

(continued from previous page)

```
    ax.plot_surface(mgx, mgy, uu, cmap=cm.plasma, rstride=1, cstride=1,␣
 ↪linewidth=0)
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_zlabel("z")

    print(uu)


demo_laplace_equation()
```

```
[[ 4. -1. -0. ... -0. -0. -0.]
 [-1.  4. -1. ... -0. -0. -0.]
 [-0. -1.  4. ... -0. -0. -0.]
 ...
 [-0. -0. -0. ...  4. -1. -0.]
 [-0. -0. -0. ... -1.  4. -1.]
 [-0. -0. -0. ... -0. -1.  4.]]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1.]
[[0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.        ]
 [0.         0.01748219 0.03486682 0.0522173  0.06989258 0.08865239
  0.1097321  0.13478052 0.16501338 0.19648103 0.19717019 0.        ]
 [0.         0.03506193 0.06976778 0.10410979 0.13870066 0.17498487
  0.21549548 0.26437659 0.32879197 0.42374056 0.59219973 0.        ]
 [0.         0.05299774 0.10503259 0.15575341 0.20581538 0.25709095
  0.31288838 0.37843838 0.46203736 0.57748951 0.74788818 0.        ]
 [0.         0.07189644 0.14161144 0.20805589 0.27171651 0.33467517
  0.4005287  0.47445121 0.56342957 0.67629194 0.82186346 0.        ]
 [0.         0.09297659 0.18146082 0.26314221 0.33831959 0.40936452
  0.48010005 0.55540819 0.64093777 0.74238522 0.86327373 0.        ]
 [0.         0.1185491  0.22811304 0.32473253 0.40905512 0.48436326
  0.55509879 0.62614372 0.70252809 0.78903744 0.88884624 0.        ]
 [0.         0.15310678 0.28770971 0.39861976 0.48880511 0.5639346
  0.62978814 0.69153981 0.75399344 0.82239021 0.9030738  0.        ]
 [0.         0.20616831 0.37099925 0.49323168 0.58361094 0.65278192
  0.70857935 0.75623394 0.79951563 0.84345617 0.90105875 0.        ]
 [0.         0.30056722 0.4968873  0.61969679 0.69962506 0.75500278
  0.79551339 0.82530099 0.84437897 0.85086008 0.85770503 0.        ]
 [0.         0.49921329 0.69628592 0.78904311 0.84018973 0.87209074
  0.89317045 0.90507766 0.90183919 0.85790014 0.67890129 0.        ]
 [1.         1.         1.         1.         1.         1.
  1.         1.         1.         1.         1.         1.        ]]
```


0
20
40
60
80
0
20
40
60
80


1.0
0.8
0.6
0.4
0.2
0.0
1.0
0.8
0.6
0.4
y
0.2
0.0
0.0
0.2
0.4
0.6
x
0.8
1.0

CHAPTER 11

Fast Fourier Transform (FFT)

```
import timeit

import matplotlib.pyplot as plt
import numpy as np
from numpy.typing import NDArray
from scipy import signal
from scipy.fft import fft, fftshift, irfft, next_fast_len, rfft, rfftfreq
```

In the notebook on interpolation, we saw that we can model data using (piecewise) polynomial basis functions. When modeling *cyclical* data, it is more informative to use trigonometric functions (sines and cosines) as basis function.

Representing a function as a linear combination of sines and cosines decomposes the function into its components of various frequencies, similarly to how a prism resolves a light beam into its constituent colors. The resulting coefficients reveal which frequencies are present in the function/data and in what amounts.

Moreover, we'll see that the representation of the function/data in *frequency domain* (as opposed to *time domain*) makes it possible to perform manipulations necessary in signal processing, or in solving differential equations in a very efficient way.

The Fourier transform in computational sciences is a vast topic by itself, with extensive specialized literature. For example, the book The Fourier transform and its applications by Ronald N. Bracewell is a classic reference.

11.1 Notation

We'll use complex exponential notation because it will allow us to write down the equations in a convenient way.

Recall **Euler's identity**

$$e^{i\theta} = \cos\theta + i\sin\theta$$

with $i = \sqrt{-1}$.

This allows us to write a cosine or sine wave with frequency f as

$$\cos(2\pi ft) = \frac{e^{2\pi ift} + e^{-2\pi ift}}{2}$$
$$\sin(2\pi ft) = \frac{e^{2\pi ift} - e^{-2\pi ift}}{2i}$$

Now let's introduce the notation

$$\omega_n = \cos(2\pi/n) - i\sin(2\pi/n) = e^{-2\pi i/n}$$

for the (or rather *one particular*) n^{th} root of unity, meaning that $\omega_n^n = 1$.

Also note the effect of complex conjugation: $\omega_n^* = \omega_n^{-1} = \omega_n^{n-1}$.

11.2 Discrete Fourier Transform

11.2.1 Definition

Given a sequence $\mathbf{x} = [x_0, \ldots, x_{n-1}]^\intercal$, its **discrete Fourier transform** (or DFT), is the sequence $\mathbf{y} = [y_0, \ldots, y_{n-1}]^\intercal$ given by

$$y_m = \sum_{k=0}^{n-1} \omega_n^{mk} x_k, \quad \forall\, m \in \{0, 1, \ldots, n-1\}$$

This can be written in matrix notation as $\mathbf{y} = \mathbf{F}_n\mathbf{x}$ where the entries of the symmetric Fourier matrix $\mathbf{F}_n$ are given by

$$\{\mathbf{F}_n\}_{mk} = \omega_n^{mk}$$

> **Example**
>
> For $n = 4$ the Fourier matrix is given by =
>
> $$\mathbf{F}_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}$$
>
> where the subscript 4 from ω_4 was dropped for visual clarity.

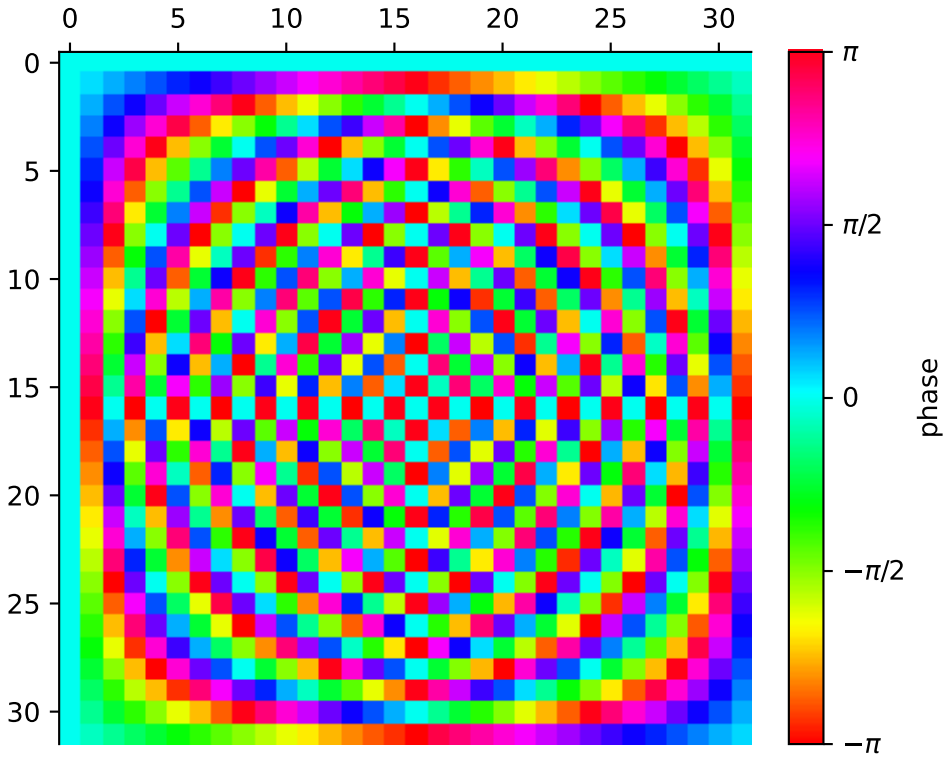
> **Visualization**
>
> The following code visualizes the Fourier matrix for $n = 32$. Since all matrix elements are complex values whose absolute value is 1, the color code just represents the phase (expressed in radians).

```
def plot_phases(n=32):
    """Visualize the phase of elements of the Fourier matrix."""
    # Make a matrix with all the phases.
    irow, icol = np.mgrid[:n, :n]
    tau = 2 * np.pi
    phase = tau * (irow * icol) / n
    phase -= np.round(phase / tau) * tau
```

(continues on next page)

(continued from previous page)

```
    # Plot the matrix.
    plt.close("phase")
    fig, ax = plt.subplots(num="phase", figsize=(5, 4))
    ms = ax.matshow(phase, cmap="hsv")
    cbar = fig.colorbar(ms)
    cbar.set_label("phase")
    cbar.set_ticks([-np.pi, -np.pi / 2, 0, np.pi / 2, np.pi])
    cbar.set_ticklabels([r"$-\pi$", r"$-\pi / 2$", "$0$", r"$\pi / 2$", r"$\pi
↪$"])
    ax.spines["top"].set_visible(True)
    ax.spines["bottom"].set_visible(False)
    ax.tick_params(axis="x", which="both", bottom=False)


plot_phases()
```



The inverse of $\mathbf{F}_n$ is given by $\mathbf{F}_n^{-1} = (1/n)\mathbf{F}_n^H$.

> **Short Proof**
>
> $$\{\mathbf{F}_n\mathbf{F}_n^{-1}\}_{mm} = \frac{1}{n}\sum_{k=0}^{n-1}\omega_n^{mk}(\omega_n^*)^{km} = 1$$
>
> $$\{\mathbf{F}_n\mathbf{F}_n^{-1}\}_{m\ell} = \frac{1}{n}\sum_{k=0}^{n-1}\omega_n^{mk}(\omega_n^*)^{k\ell} = \frac{1}{n}\sum_{k=0}^{n-1}\omega_n^{(m-\ell)k} = \frac{1}{n}\frac{1-\omega_n^{n(m-\ell)}}{1-\omega_n^{m-\ell}} = 0$$

> In the last step, we made use of $\omega_n^n = 1$.

The **inverse** DFT thus reads

$$\begin{aligned} x_k &= \frac{1}{n}\sum_{m=0}^{n-1} \omega_n^{-km} y_m \\ &= \frac{1}{n}\sum_{m=0}^{n-1} y_m \big[\cos(2\pi mk/n) + i\sin(2\pi mk/n)\big], \quad \forall\, k \in \{0, 1, \dots, n-1\} \end{aligned}$$

This inverse transform represents the components of **x** as a linear combination of sines and cosines, with coefficients given by the components of **y**. Thus, if the components of **x** are sample values of a function, then the DFT solves the trigonometric interpolation problem with nothing more than a matrix-vector multiplication.

The cost scales as $\mathcal{O}(n^2)$ instead of $\mathcal{O}(n^3)$ when solving a linear system, as we saw in the interpolation notebook. Better still, using the **Fast Fourier transform** algorithm, this can be reduced even further to $\mathcal{O}(n\log(n))$.

11.2.2 Frequencies

In the definitions of the (inverse) DFT, the frequency is represented by a dimensionless integer index m, which seems to have the wrong unit. It is implicitly assumed that the samples x_k are taken at regular (time) steps

$$t_k = t_0 + k/f_s.$$

where f_s is called the sampling rate. Put differently,

$$k = (t_k - t_0) f_s.$$

With that choice, the sine and cosine basis functions, can be written as a function of time:

$$\begin{aligned} \omega_n^{-mk} &= \cos(2\pi mk/n) + i\sin(2\pi mk/n) \\ &= \cos\left(2\pi \frac{f_s m}{n}(t_k - t_0)\right) + i\sin\left(2\pi \frac{f_s m}{n}(t_k - t_0)\right) \end{aligned}$$

As soon as a sampling rate f_s is assumed, the index m corresponds to a genuine frequency $\frac{f_s m}{n}$.

Note that one may also sample in a spatial domain instead of the time domain, in which case the index m corresponds to a wavenumber instead of a frequency.

11.2.3 Negative frequencies

One classical source of confusion is the terminology of *negative frequency* when interpreting a DFT. This seems wrong because the inverse DFT only makes use of a positive index m, such that all cosine and sine functions have positive frequencies:

$$\begin{aligned} x_k &= \frac{1}{n}\sum_{m=0}^{n-1} \omega_n^{-km} y_m \\ &= \frac{1}{n}\sum_{m=0}^{n-1} y_m \big[\cos(2\pi mk/n) + i\sin(2\pi mk/n)\big], \quad \forall\, k \in \{0, 1, \dots, n-1\} \end{aligned}$$

However, one can always make use of $\omega_n^{-mk} = \omega_n^{(\ell n - m)k}$, where ℓ is an arbitrary integer, because $\omega_n^n = 1$. With this, the inverse DFT can be rewritten in many different froms:

$$x_k = \frac{1}{n}\sum_{m=0}^{n-1} y_m \omega_n^{(n-m)k}, \quad \forall\, k \in \{0, 1, \dots, n-1\}$$

or

$$x_k = \frac{1}{n}\sum_{m=0}^{n-1} y_m \omega_n^{(-n-m)k}, \quad \forall\, k \in \{0, 1, \ldots, n-1\}$$

etc. This shows that the coefficient m is not associated with a single frequency, but rather with an infinite number of them.

For practical interpretations, one limits the infinite possibilities to those frequencies (either with positive or negative sign) with the smallest absolute value. The sequence of indexes m is

$$[0, \ldots, n-1]$$

and corresponding frequencies for interpretative purposes, assuming a sampling rate f_s, are

$$\left[0, \frac{f_s}{n}, \frac{2f_s}{n}, \ldots, \frac{\lfloor n/2 \rfloor f_s}{n}, \frac{(\lfloor n/2 \rfloor - n) f_s}{n}, \ldots, \frac{-2f_s}{n}, \frac{-f_s}{n},\right]$$

where $\lfloor \cdot \rfloor$ represents the floor function. It rounds towards a lower (more negative) integer. In SciPy, the function `fftfreq` constructs this type of frequency axis.

If you have trouble with the meaning of a negative frequency, think of positive frequencies as *clockwise* and negative frequencies as *counterclockwise*.

> **Illustration with sinusoidal input**
>
> The following code cell illustrates basic DFT concepts. The input for the DFT is a sinusoidal function, of which you can control the frequency, phase, sampling rate and number of samples.

```
def compute_fft_demo_data(
    freq0: float, sampling_rate: float, phase: float, nsample: int
) -> tuple[NDArray, NDArray, NDArray, NDArray]:
    """Generate a sinusoidal signal and its FFT.

    Parameters
    ----------
    freq0
        Frequency of the sinusoidal signal.
    sampling_rate
        Sampling rate.
    phase
        Phase shift of the signal.
    nsample
        Number of samples.

    Returns
    -------
    time_axis
        Time axis.
    signal_time
        Sinusoidal signal.
    freq_axis
        Frequency axis.
    signal_freq
        Discrete Fourier Transform of the signal.
    """
```

(continues on next page)

(continued from previous page)

```
    # All values are in SI base units.
    # This example is put in a function to avoid that
    # the local variables interfere with the examples
    # below.

    # 'Sample' a sine signal
    # and compute its DFT.
    time_axis = np.arange(nsample) / sampling_rate
    signal_time = np.sin(2 * np.pi * freq0 * time_axis + phase)
    signal_freq = fft(signal_time)

    # For the sake of plotting the FFT, construct a frequency
    # axis in which we make some frequencies explicitly negative.
    freq_axis = np.arange(nsample) * sampling_rate / nsample
    freq_axis[nsample // 2 + 1 :] -= sampling_rate
    # One might also use
    # freq_axis = fftfreq(nsample, 1/sampling_rate)

    return time_axis, signal_time, freq_axis, signal_freq


def fft_demo(sampling_rate, freq0, phase, nsample, num):
    time_axis, signal_time, freq_axis, signal_freq = compute_fft_demo_data(
        freq0, sampling_rate, phase, nsample
    )
    plt.close(num)
    fig, (ax0, ax1) = plt.subplots(1, 2, figsize=(8, 4), num=num, clear=True)

    # Create sine signal for reference (in meters as a function of time in␣
↪seconds)
    # Sine signal is plotted till last sample point
    t = np.linspace(0, time_axis[-1], 10000)

    # Plot the sine signal and sample functions
    ax0.plot(
        t, np.sin(2 * np.pi * freq0 * t + phase), "-", color="blue", label=
↪"Signal"
    )
    ax0.scatter(
        time_axis, signal_time, color="red", label="Sample Points", s=20,␣
↪zorder=5
    )
    ax0.plot(time_axis, signal_time, ":", color="red", label="Sample Output")
    ax0.set_xlim(0, 5)
    ax0.set_xlabel("Time [s]")
    ax0.set_ylabel("Time-dependent position [m]")
    ax0.legend(loc="upper right")

    # Create a list of all frequency values (used in set_ylim)
    all_freq = signal_freq.real + signal_freq.imag

    # Plot the DFT
    ax1.scatter(freq_axis, signal_freq.real, s=15, label="Real part",␣
↪zorder=5)
```

(continues on next page)

(continued from previous page)

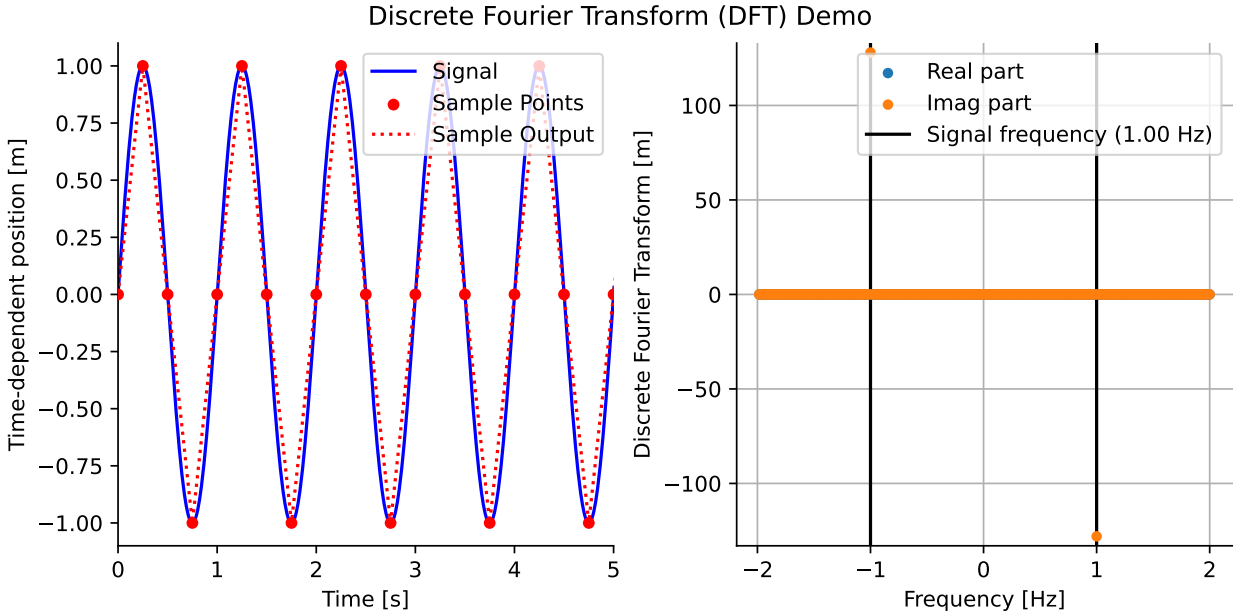
```
    ax1.scatter(freq_axis, signal_freq.imag, s=15, label="Imag part",␣
↪zorder=5)
    ax1.axvline(-freq0, color="k", label=f"Signal frequency ({freq0:.2f} Hz)")
    ax1.axvline(freq0, color="k")
    ax1.set_xlabel("Frequency [Hz]")
    ax1.set_ylabel("Discrete Fourier Transform [m]")
    ax1.set_ylim([min(all_freq) - 5, max(all_freq) + 5])
    ax1.legend()
    ax1.grid()

    fig.suptitle("Discrete Fourier Transform (DFT) Demo")
```

Let's first consider a sine input. It can be decomposed as:

$$\sin(2\pi f t) = \frac{i}{2}\exp(-i2\pi f t) - \frac{i}{2}\exp(+i2\pi f t)$$

This corresponds to two imaginary non-zero values in the descrete Fourier transform.

```
fft_demo(sampling_rate=4, freq0=1, phase=0, nsample=256, num="case1_sine")
```

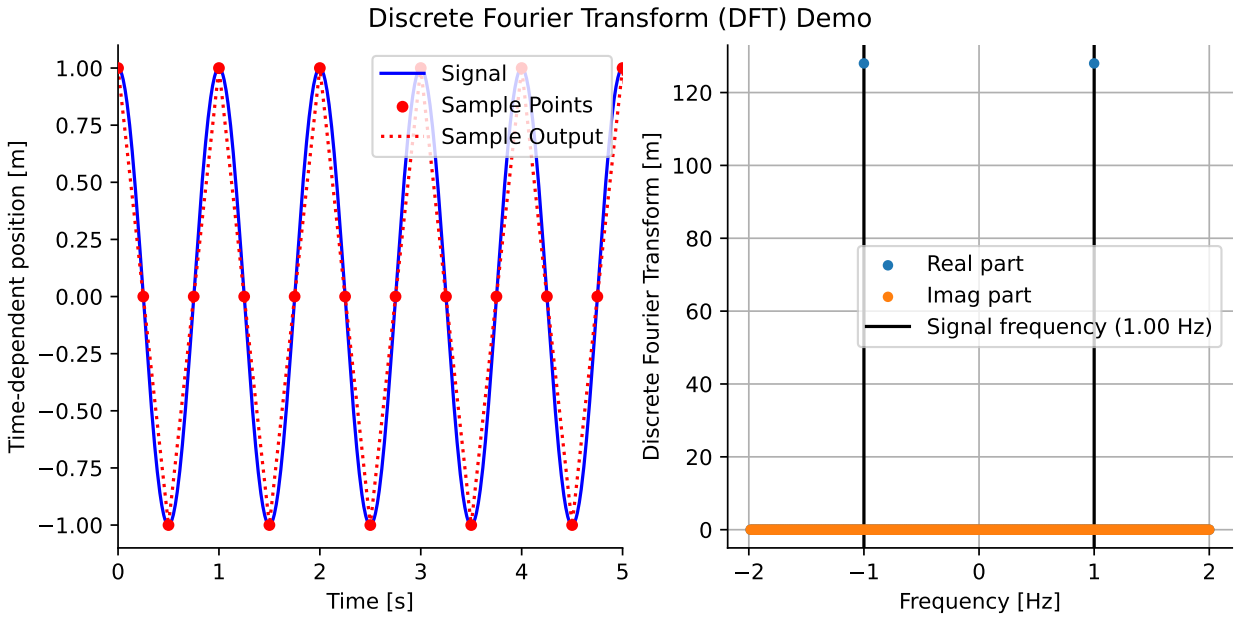


We can also look at the result for a similar cosine, which can be decomposed as:

$$\cos(2\pi f t) = \frac{1}{2}\exp(-i2\pi f t) + \frac{1}{2}\exp(+i2\pi f t)$$

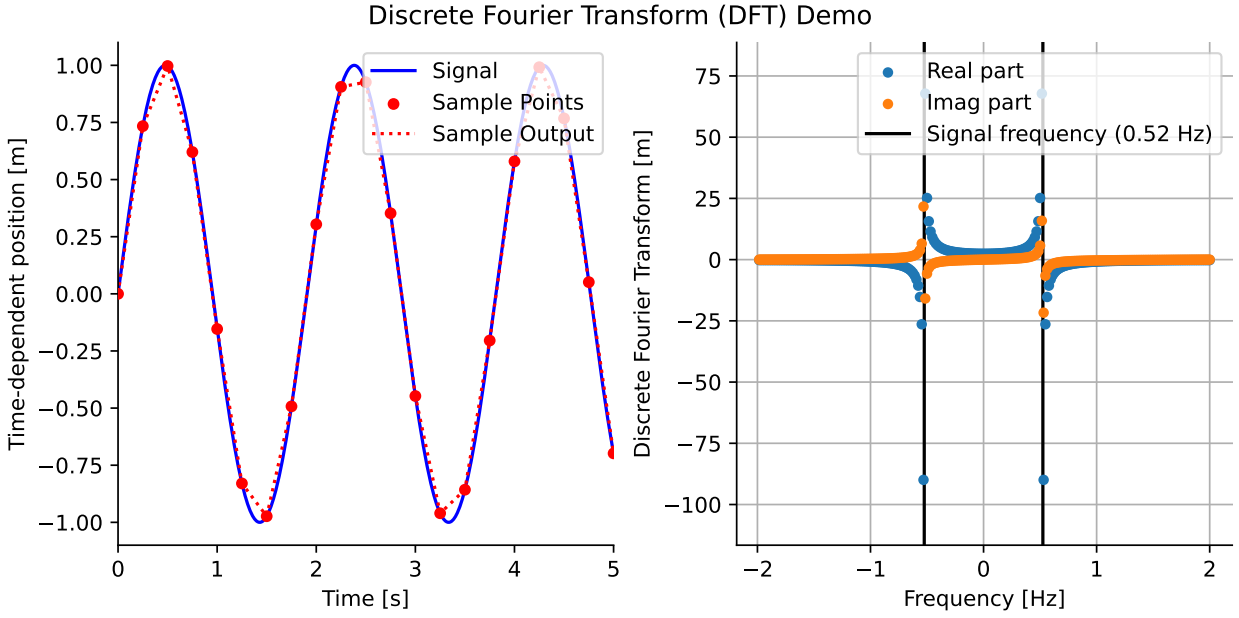
This corresponds to two identical real non-zero values in the descrete Fourier transform.

```
fft_demo(sampling_rate=4, freq0=1, phase=np.pi / 2, nsample=256, num="case2_
↪cosine")
```



Finally, we can analyze a sine function with a fractional number of periods in the sequence. The DFT has many non-zero values around the expected frequency. This phenomenon is called spectral leakage.

```
fft_demo(sampling_rate=4, freq0=0.5246, phase=0, nsample=256, num="case3_
↪leakage")
```



Note that the DFT is computed with the FFT algorithm from SciPy. This algorithm will be explained in the next section, but how it works is not important for this example. For now, only the result matters.

11.2.4 Nyquist frequency & Nyquist-Shannon sampling theorem

In the sequence of frequencies shown above, the highest one is $\lfloor n/2 \rfloor f_s/n$. In the limit of many samples, or for any even number of samples, the highest frequency always becomes $f_s/2$, which is known as the Nyquist frequency.

If the underlying time-dependent function, of which x_k are samples, contains relevant fluctuations at frequencies above $f_s/2$, it will be impossible to discern them from lower-frequency fluctuations. The reason is that, on the sampling grid, the basis function ω_n^{mk} is indistinguishable from $\omega_n^{(\ell n+m)k}$, where ℓ is an arbitrary integer.

In practice, this means that the sampling frequency should at least twice the highest relevant frequency of the underlying function. This is known as the Nyquist-Shannon sampling theorem.

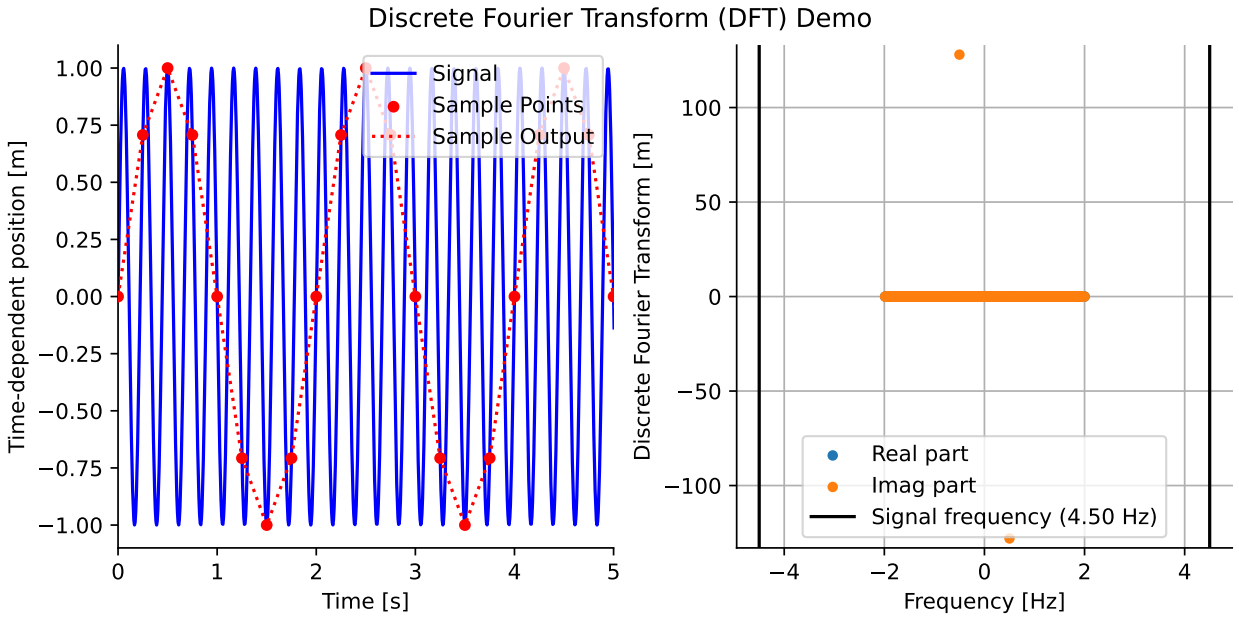
> **Example**
>
> As shown by the `scipy` example below, the DFT of the sequence $[+1\ -1\ +1\ -1\ +1\ -1\ +1\ -1]^\mathsf{T}$, which has the highest possible rate of oscillation is given by
>
> $$\mathbf{F}_8\mathbf{x} = \mathbf{F}_8 \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \\ 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 8 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$
>
> In the transformed sequence, we see only a nonzero component at the Nyquist frequency (y_4), and y_0 indeed equals the sum of the elements of $\mathbf{x}$.

```
fft([1, -1, 1, -1, 1, -1, 1, -1])
```

```
array([0.-0.j, 0.+0.j, 0.-0.j, 0.+0.j, 8.-0.j, 0.-0.j, 0.+0.j, 0.-0.j])
```

> **Illustration with the `fft_demo` function (Nyquist & Shannon)**
>
> In the following cell, the `fft_demo` is executed with a frequency $f_0 = f_s + 0.5$. In line with the theory, the results look identical to the case of $f_0 = 0.5$. Try to see what happens when $f_0 = f_s/2 + 0.5$. Can you explain the result?

```
fft_demo(sampling_rate=4, freq0=4.5, phase=0, nsample=256, num="case4_nyquist
↪")
```



11.2.5 The DC component (zero frequency)

The lowest of all frequencies is always zero (in Hz or m^{-1}). The corresponding coefficient of the DFT is simply the sum of all values x_k and is called the DC (direct current) component.

Note that a constant shift of all x_k will only affect the DC component.

> **Simple illustration**
>
> The following two inputs to the `fft` function only differ by a constant shift, which is reflected in (only) a difference in DC component.

```
fft([2, 0, 2, 0, 2, 0, 2, 0])
```

```
array([8.-0.j, 0.+0.j, 0.-0.j, 0.+0.j, 8.-0.j, 0.-0.j, 0.+0.j, 0.-0.j])
```

```
fft([1, -1, 1, -1, 1, -1, 1, -1])
```

```
array([0.-0.j, 0.+0.j, 0.-0.j, 0.+0.j, 8.-0.j, 0.-0.j, 0.+0.j, 0.-0.j])
```

11.2.6 DFT of a real sequence

The DFT of a sequence (even a real sequence) is in general complex. This is not something to worry about, as the inverse DFT will take us back to the real domain.

The DFT of a real sequence of length n has $2n$ real and imaginary parts, but still contains only n independent pieces of information. This can be understood as follows: for each basis function ω_n^{-km}, there is another basis function $\omega_n^{k(n+m)} = \omega_n^{km} = (\omega^*)_n^{-km}$, which is just its complex conjugate. To represent a real sequence $\mathbf{x}$, the DFT must adhere to $y_m = y_{n-m}^*$.

Short Proof

$$y^*_{n-m} = \left(\sum_{k=0}^{n-1} \omega_n^{k(n-m)} x_k\right)^* = \left(\sum_{k=0}^{n-1} \omega_n^{kn}\omega_n^{-km} x_k\right)^* = \left(\sum_{k=0}^{n-1} \omega_n^{-km} x_k\right)^*$$
$$= \sum_{k=0}^{n-1} (\omega_n^*)^{-km} x_k^* = \sum_{k=0}^{n-1} \omega_n^{km} x_k^* = \sum_{k=0}^{n-1} \omega_n^{km} x_k = y_m$$

In the third step, we made use of $\omega_n^{kn} = 1$.

This has a few implications for real sequences:

- y_0 is real as it is just equal to the sum of the real components x_k.
- When n is even, $y_{n/2}$ is also real.
- When n is odd, $y_{(n+1)}/2$ may still be complex.

Because the second half of a DFT of a real sequence is redundant, a specialized function `rfft` exists to compute only the first half of the DFT.

Simple illustration

The following uses as input a real sequence, and as expected, the last part of the DFT contains redundant information.

```
fft([1.2, 2.5, 3.1, 2.3, 1.4, 1.0])
```

```
array([11.5-0.j        , -1.6-2.77128129j, -0.5+0.17320508j,
       -0.1-0.j        , -0.5-0.17320508j, -1.6+2.77128129j])
```

The `rrft` function only accepts real inputs and is faster because it skips the redundant computations for real inputs.

```
rfft([1.2, 2.5, 3.1, 2.3, 1.4, 1.0])
```

```
array([11.5+0.j        , -1.6-2.77128129j, -0.5+0.17320508j,
       -0.1+0.j        ])
```

The following illustrates that the last element of the `rfft` function corresponds to the Nyquist frequency.

```
rfft([1, -1, 1, -1, 1, -1, 1, -1])
```

```
array([0.+0.j, 0.+0.j, 0.-0.j, 0.+0.j, 8.+0.j])
```

11.3 FFT Algorithm

We can exploit certain symmetries and redundancies in the definition of the DFT to calculate it efficiently, with a cost scaling like $\mathcal{O}(n\log(n))$ instead of $\mathcal{O}(n^2)$. This will first be illustrated with an example for $n = 4$:

Example

From the definition of the DFT, we have

$$y_m = \sum_{k=0}^{3} x_k \omega_4^{mk}, \quad m = 0, \ldots, 3$$

Again, for the sake of visual clarity, we will use $\omega = \omega_4$ below.

Writing this out in full, we have

$$\begin{aligned}
y_0 &= x_0\omega^0 + x_1\omega^0 + x_2\omega^0 + x_3\omega^0 \\
y_1 &= x_0\omega^0 + x_1\omega^1 + x_2\omega^2 + x_3\omega^3 \\
y_2 &= x_0\omega^0 + x_1\omega^2 + x_2\omega^4 + x_3\omega^6 \\
y_3 &= x_0\omega^0 + x_1\omega^3 + x_2\omega^6 + x_3\omega^9
\end{aligned}$$

We know that

$$\begin{aligned}
\omega^0 &= \omega^4 = 1 \\
\omega^2 &= \omega^6 = -1 \\
\omega^9 &= \omega^1
\end{aligned}$$

so we can regroup the equations as

$$\begin{aligned}
y_0 &= (x_0 + x_2) + \omega^0(x_1 + x_3) \\
y_1 &= (x_0 - x_2) + \omega^1(x_1 - x_3) \\
y_2 &= (x_0 + x_2) + \omega^2(x_1 + x_3) \\
y_3 &= (x_0 - x_2) + \omega^3(x_1 - x_3)
\end{aligned}$$

This shows that computing the DFT of the original 4 point sequence has been reduced to computing the DFT of its 2 point even and odd subsequences:

$$\begin{aligned}
\tilde{y}_0^{\text{even}} &= x_0 + \omega_2^0 x_2 \\
\tilde{y}_1^{\text{even}} &= x_0 + \omega_2^1 x_2 \\
\tilde{y}_0^{\text{odd}} &= x_1 + \omega_2^0 x_3 \\
\tilde{y}_1^{\text{odd}} &= x_1 + \omega_2^1 x_3
\end{aligned}$$

and

$$\begin{aligned}
y_0 &= \tilde{y}_0^{\text{even}} + \omega^0 \tilde{y}_0^{\text{odd}} \\
y_1 &= \tilde{y}_1^{\text{even}} + \omega^1 \tilde{y}_1^{\text{odd}} \\
y_2 &= \tilde{y}_0^{\text{even}} + \omega^2 \tilde{y}_0^{\text{odd}} \\
y_3 &= \tilde{y}_1^{\text{even}} + \omega^3 \tilde{y}_1^{\text{odd}}
\end{aligned}$$

This property holds in general: the DFT of an n-point sequence can be computed by breaking it into two DFTs of half the length, provided n is even.

Example (continued)

In matrix notation, this becomes more clear. Consider the first few Fourier matrices:

$$\mathbf{F}_1 = 1$$

$$\mathbf{F}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$\mathbf{F}_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}$$

Additionally let $\mathbf{P}_4$ be the permutation matrix, which separates the odd and even subsequences:

$$\mathbf{P}_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and $\mathbf{D}_2$ the diagonal matrix

$$\mathbf{D}_2 = \text{diag}(1, \omega_4^1) = \begin{bmatrix} 1 & 0 \\ 0 & -i \end{bmatrix}$$

With these matrices, we can rearrange $\mathbf{F}_4$ such that each block is a diagonally scaled version of $\mathbf{F}_2$.

$$\mathbf{F}_4\mathbf{P}_4 = \left[\begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 1 & -1 & -i & i \\ \hline 1 & 1 & -1 & -1 \\ 1 & -1 & i & -i \end{array}\right] = \begin{bmatrix} \mathbf{F}_2 & \mathbf{D}_2\mathbf{F}_2 \\ \mathbf{F}_2 & -\mathbf{D}_2\mathbf{F}_2 \end{bmatrix}$$

In general, $\mathbf{P}_n$ is the permutation that groups the even-numbered columns of $\mathbf{F}_n$ before the odd numbered columns and

$$\mathbf{D}_{n/2} = \text{diag}\left(1, \omega_n, \ldots, \omega_n^{(n/2)-1}\right)$$

Thus, to apply $\mathbf{F}_n$ to a sequence of length n, we merely need to apply $\mathbf{F}_{n/2}$ to its even and odd subsequences and scale the results by $\pm\mathbf{D}_{n/2}$. This recursive *divide-and-conquer* approach is called the **fast Fourier transform** and can be used to recursively calculate the DFT of sequences of any length (that is a power of 2).

Despite its name, the fast Fourier transform is an algorithm and not a transform. It's a specific way of efficiently calculating the discrete Fourier transform. There are $\log_2(n)$ levels of recursion, each of which involves $\mathcal{O}(n)$ operations, so the total cost of the FFT scales as $\mathcal{O}(n\log(n))$. The following table shows that, for large sequences, this makes a huge practical difference compared to the $\mathcal{O}(n^2)$ required for a straightforward implementation of the DFT:

n	$n\log_2(n)$	n^2
8	24	64
16	64	256
32	160	1024
64	384	4096
128	896	16384
256	2048	65536
512	4608	262144
1024	10240	1048576

History

The reason why scientists started working on finding a method to calculate Fourier Transforms more efficiently

> with less operations goes back to the aftermath of World War II. After the USA used their atomic bomb to shock the whole world, a lot of other countries started testing nuclear weapons for further use. After a convention in Geneva the big nations agreed to ban further testing in the atmosphere and under water, but not underground. Why? It's hard to detect if an atomic bomb is released underground, but by decomposing seismic signals it can be analyzed! Sadly the discovery of the FFT algorithm was too late to prevent an all out nuclear race. This means if it was invented earlier the Cold War maybe never happened!!
>
> Source: *Veritasium. (2022a, november 3). The Remarkable Story Behind The Most Important Algorithm Of All Time Youtube Video.
>
> Since its discovery by Cooley and Tukey in 1965, the practical value of the Fast Fourier Transform algorithm has been immense for many fields of research and applications. However, it was already discovered by Carl Friedrich Gauss 160 years earlier, but somehow was forgotten again!

11.3.1 Limitations and extensions

Although very efficient for some use cases, the FFT algorithm has a few limitations:

- The input sequence needs to be **equally spaced** (although this sometimes can be mitigated using interpolation).
- The input sequence is assumed to be **periodic** (resulting from the definition of the DFT, which tries to transform the data in a linear combination of sines and cosines).

  > In signal processing, the discrete cosine transform (DCT) is often used instead of DFT. It does not assume the input is periodic, making it more broadly applicable. Furthermore, the DCT of a real signal is also real, making the output more intuitive. The DCT is also implemented in SciPy, with algorithms very similar to the FFT.

- The sequence is assumed to be a **power-of-2 in length** (specific to our algorithm).

  > The FFT can be generalized to handle sequences of arbitrary length, not only powers of 2. The general FFT algorithm does not only split a sequence into two. Instead, it partitions a sequence in $M \geq 2$ subsets, where M is the smallest prime factor of the length of a sequence, at the current recursion level. So in general, M may vary across different levels of recursion. At each level, an M-fold DFT must be computed with conventional matrix-vector multiplication.
  >
  > This shows that the FFT, as discussed here, will be slow for long sequences whose length is a prime number. For such cases, more advanced implementations exist to efficiently handle sequences whose lengths are prime numbers. These advanced algorithms still result in an $\mathcal{O}(n \log(n))$ scaling, but with a much larger prefactor, compared to the algorithm discussed here.

11.3.2 Windowing (elective, can be skipped)

Windowing is employed to address a specific challenge associated with FFT: spectral leakage. This phenomenon occurs when analyzing finite segments of a signal, causing inaccuracies in frequency domain representation. For an overview of various windowing functions, you can explore Window function on Wikipedia.

Mathematically, applying a window involves multiplying the original signal by a window function, w_k. One common window function is the cosine window, expressed as:

$$w_k = \frac{1}{2} - \frac{1}{2}\cos\left(\frac{2\pi k}{n-1}\right)$$

Before applying the discrete Fourier transform, the signal is first multiplied with the window:

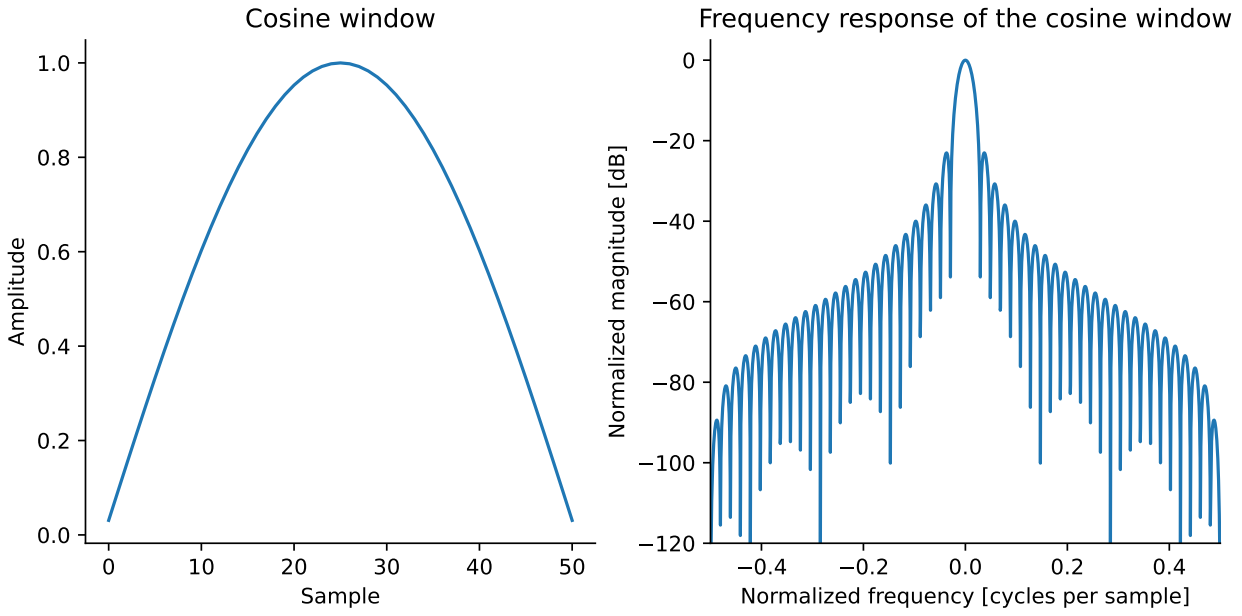
$$\tilde{x}_k = w_k x_k$$

Here, $\tilde{x}_k$ represents the windowed signal, k is the sample index, and n is the total number of samples. The window function gently tapers the signal at its edges, reducing abrupt changes and mitigating spectral leakage.

Windowing offers numerous benefits, including reduced spectral leakage, improved frequency resolution, and control over side lobe suppression. The choice of a specific window function depends on the requirements of the signal processing task at hand and the characteristics desired in the frequency domain analysis.

The following example shows a typical plot of a window function. To characterize the window function, it can be Fourier transformed as such, which effectively shows the result of windowing a constant function. The Fourier transform shows a central peak at frequency zero (corresponding to the constant signal), which has a certain width and side lobes due to the fact that the function is truncated. When the `cosine` window by another function from `scipy.signal.windows`, the shape of the main peak and the side lobes in the spectrum will change.

```
def demo_windowing():
    window = signal.windows.cosine(51)
    # window = signal.windows.blackman(51)
    # window = signal.windows.bartlett(51)

    plt.close("windowing")
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4), num="windowing")
    ax1.plot(window)
    ax1.set_title("Cosine window")
    ax1.set_ylabel("Amplitude")
    ax1.set_xlabel("Sample")

    A = fft(window, 2047) / (len(window) / 2.0)
    freq = np.linspace(-0.5, 0.5, len(A))
    response = 20 * np.log10(np.abs(fftshift(A / abs(A).max())))
    ax2.plot(freq, response)
    ax2.axis([-0.5, 0.5, -120, 5])
    ax2.set_title("Frequency response of the cosine window")
    ax2.set_ylabel("Normalized magnitude [dB]")
    ax2.set_xlabel("Normalized frequency [cycles per sample]")


demo_windowing()
```



11.3.3 Performance demo

In the following cells, we're going to perform the FFT of a sequence with lengths ranging from 1 to 4100, and track the required CPU time. Next, we're going to repeat this, but now by zero-padding each sequence until the length given by `next_fast_len`.

In the resulting figures, you'll see that powers of 2 are indeed the most efficient, and that prime numbers are the least efficient lengths, even for the optimized `scipy` implementation. However, zero padding the input until `next_fast_len` mitigates part of these problems (although the zero padding in itself might introduce spurious noise in your Fourier transform!)

Sequence of prime numbers

We'll first make a little detour and show how to construct a sequence of prime numbers using vectorization techniques in NumPy. The algorithm is based on Eratosthenes' sieve.

```
def construct_primes(nabove: int) -> np.ndarray:
    """Return an array with prime numbers below ``nabove``."""
    # This is the Python equivalent of the sieve of Eratosthenes.
    sieve = np.ones(nabove, bool)
    sieve[:2] = False
    divisor = 2
    maxdivisor = np.sqrt(nabove)
    while divisor < maxdivisor:
        if sieve[divisor]:
            sieve[2 * divisor :: divisor] = False
        divisor += 1
    return sieve.nonzero()[0]


assert (construct_primes(10) == [2, 3, 5, 7]).all()
print(construct_primes(50))
```

```
[ 2  3  5  7 11 13 17 19 23 29 31 37 41 43 47]
```

Timing function

The following cell implements a function that times the `fft` function with random input data for a series of lengths.

```
def time_fft(maxlen: int, pad: bool = False) -> np.ndarray:
    """Time the fft function for a series of input sizes.

    Parameters
    ----------
    maxlen
        The maximum sequence length (inclusive).
    pad
        When set to True, input arrays will be zero-padded.
        The total size is determined by scipy.fft.next_fast_len.

    Returns
    -------
    timings
        An array with timings for input sizes 1 to maxlen.
    """
    rng = np.random.default_rng(1)
    data_maxlen = rng.uniform(0, 1, maxlen) + 1.0j * rng.uniform(0, 1, maxlen)
    timings = np.zeros(maxlen)
```

(continues on next page)

(continued from previous page)

```
    for length in range(1, maxlen + 1):
        # Define a minimal namespace for the timeit function,
        namespace = {
            "data": data_maxlen[:length],
            "fft": fft,
            "length": next_fast_len(length) if pad else length,
        }
        timing = timeit.timeit("fft(data, length)", number=10,␣
↪globals=namespace)
        timings[length - 1] = timing
    return timings


assert (time_fft(10) < 0.1).all()
assert (time_fft(10, True) < 0.1).all()
```

Plotting function

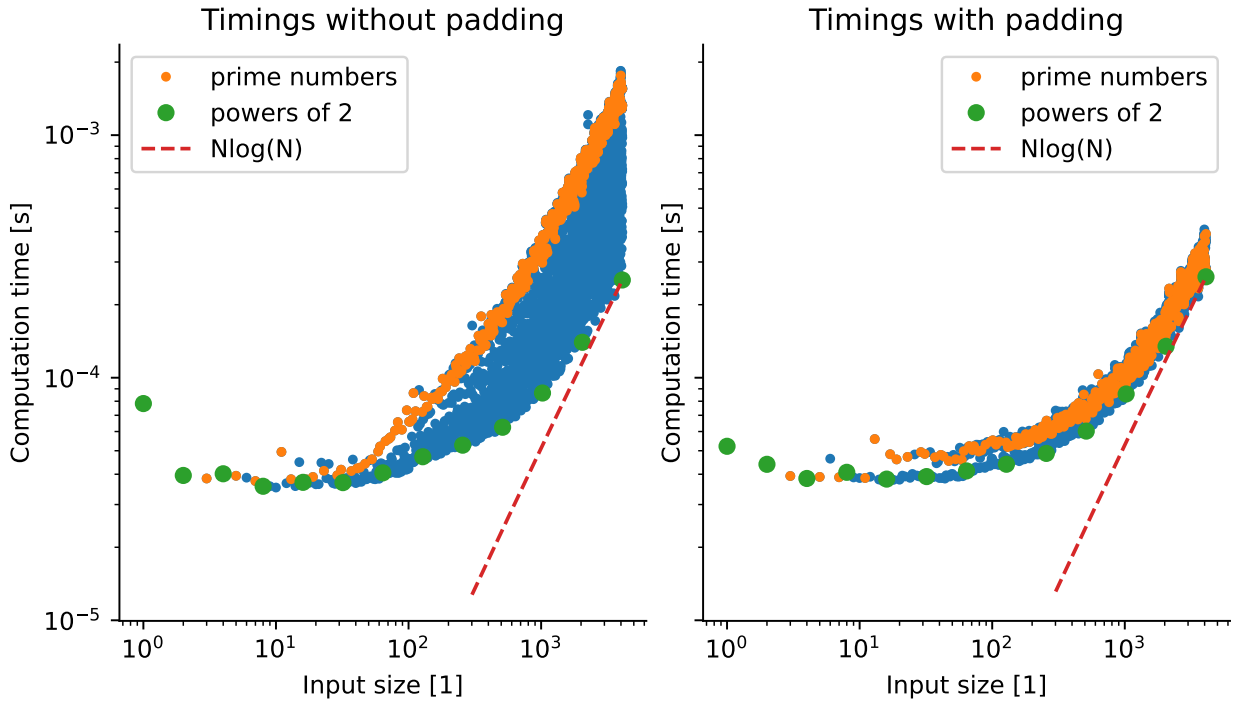
The following cell uses the functions `construct_primes` and `time_fft` to plot the cost of `fft` as a function of input size.

```
def plot_timings_fft(maxlen):
    """Plot the timing of the FFT algorithm as a function of input size.

    Parameters
    ----------
    maxlen
        The maximum input size to consider.

    """
    sizes = np.arange(1, maxlen + 1)
    timings_normal = time_fft(maxlen, False)
    timings_padded = time_fft(maxlen, True)

    primes = construct_primes(maxlen)
    powers2 = 2 ** np.arange(int(np.floor(np.log2(maxlen))) + 1)

    plt.close("timing_fft")
    fig, axs = plt.subplots(1, 2, figsize=(7, 4), num="timing_fft",␣
↪sharey=True)

    for ax, timings in zip(axs, [timings_normal, timings_padded],␣
↪strict=False):
        ax.plot(sizes, timings, ".")
        ax.plot(primes, timings[primes - 1], ".", label="prime numbers")
        ax.plot(powers2, timings[powers2 - 1], "o", label="powers of 2")
        last_p2 = powers2[-1]
        scale = timings[last_p2 - 1] / (last_p2 * np.log2(last_p2))
        ax.plot(
            sizes[300:],
            sizes[300:] * np.log2(sizes[300:]) * scale,
            "--",
            label="Nlog(N)",
        )
        ax.set_xlabel("Input size [1]")
```

(continues on next page)

(continued from previous page)

```
        ax.set_ylabel("Computation time [s]")
        ax.set_xscale("log")
        ax.set_yscale("log")
        ax.legend()
    axs[0].set_title("Timings without padding")
    axs[1].set_title("Timings with padding")


plot_timings_fft(4100)
```



11.4 Applications

In many cases, the spectrum (in frequency domain) of a dataset is the quantity of interest, for which FFT is an obvious choice. However, there are many more applications that, sometimes unexpectedly, rely on the Fast Fourier Transform.

11.4.1 Signal processing

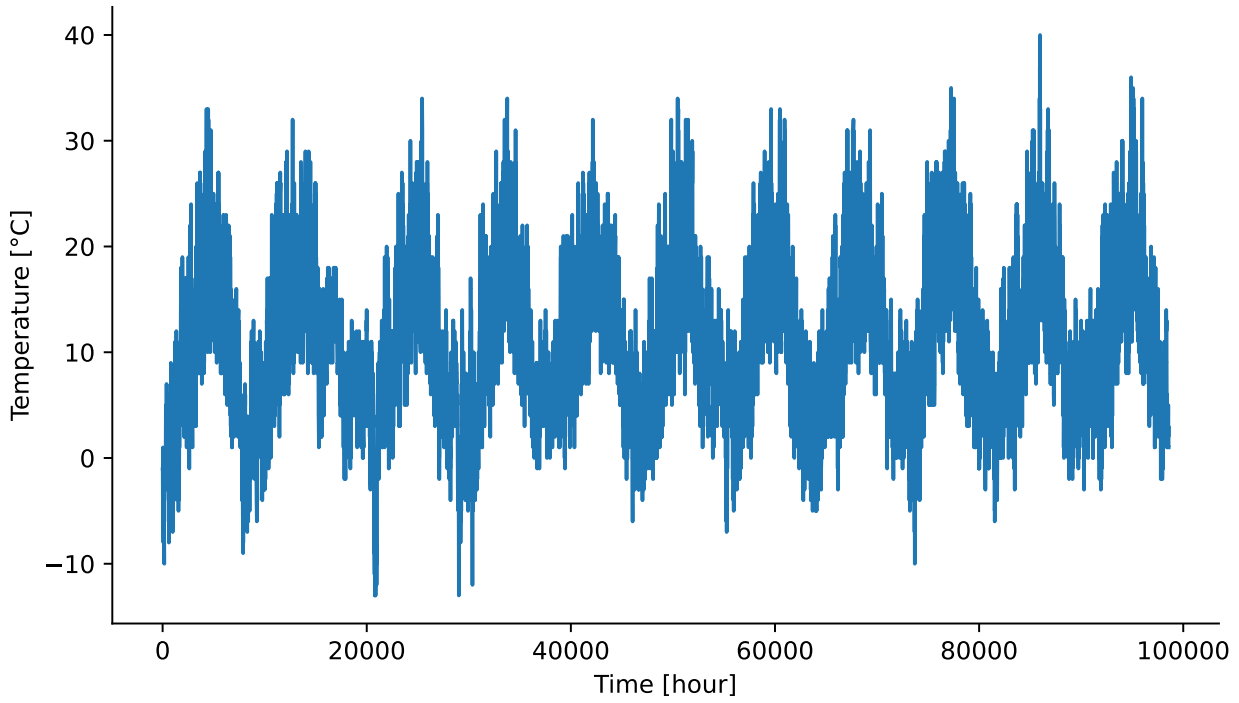
Signal processing is one of the domains where the DFT is used to manipulate streams of data. For example, to filter out unwanted high-frequency noise from measurement data, one can take the FFT, set all high-frequency components to zero and then take the inverse transform.

Another example is the processing of data which contains multiple periodicities, which you want to investigate separately of each other. E.g. a climate researcher might be interested in daily temperature fluctuations and wants to investigate them in the absence of the yearly cycle that lies on top of this data. In such case, unwanted cycles can easily be removed in frequency domain, as shown in the example below.

> **Example**
>
> The file `fft_weather.txt` contains hourly temperature data from Brussels airport from January 2010 until January 2021. (source: https://mesonet.agron.iastate.edu/request/download.phtml?network=BE__ASOS) We

> are interested in investigating yearly or daily temperature fluctuations, independently from each other.
>
> First, let's see what the data looks like and whether we can identify these expected cycles.

```
def plot_temperature():
    """Show the raw temperature data."""
    data = np.loadtxt("fft_weather.txt")
    plt.close("raw_temp")
    fig, ax = plt.subplots(num="raw_temp", figsize=(7, 4))
    ax.plot(data)
    ax.set_xlabel("Time [hour]")
    ax.set_ylabel("Temperature [°C]")


plot_temperature()
```

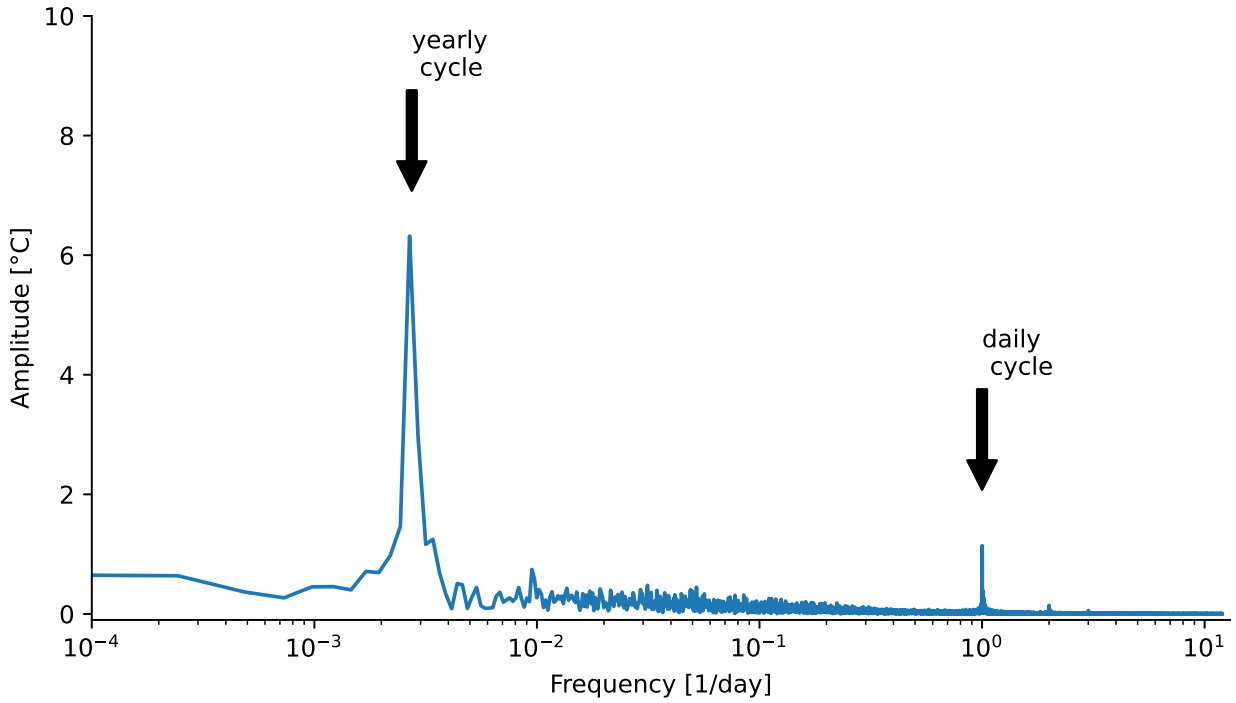



```
def fft_temperature():
    """Visualize the spectrum of the temperature changes."""
    # Load the data and compute the FFT.
    data = np.loadtxt("fft_weather.txt")
    spectrum = rfft(data)
    freqs = rfftfreq(len(data), 1 / 24)

    # Plot absolute value of the DFT.
    # The amplitude is divided by len(spectrum) before plotting
    # to show the amplitude of the fluctuations on an
    # intuitive scale.
    plt.close("first_spectrum")
    fig, ax = plt.subplots(figsize=(7, 4), num="first_spectrum")
    ax.annotate(
        "daily\n cycle",
        xy=(1, 2),
```

(continues on next page)

(continued from previous page)

```
        xytext=(1, 4),
        arrowprops=dict(facecolor="black", shrink=0.04),
    )
    ax.annotate(
        "yearly\n cycle",
        xy=(1 / 365.0, 7),
        xytext=(1 / 365, 9),
        arrowprops=dict(facecolor="black", shrink=0.04),
    )
    ax.plot(freqs, abs(spectrum) / len(spectrum), "-")
    ax.set_xlim(1e-4, 13)
    ax.set_ylim(-0.1, 10)
    ax.set_xscale("log")
    ax.set_xlabel("Frequency [1/day]")
    ax.set_ylabel("Amplitude [°C]")


fft_temperature()
```



> Now, let's try to get rid of the short-term fluctuations, such that the longer-term trends can be investigated more clearly. We'll set all frequency components above 100 days to zero.

```
def filter_slow():
    """Retain only the slow fluctuations and plot the result."""
    # Load the data and compute the FFT.
    data = np.loadtxt("fft_weather.txt")
    spectrum = rfft(data)
    freqs = rfftfreq(len(data), 1 / 24)
```

(continues on next page)

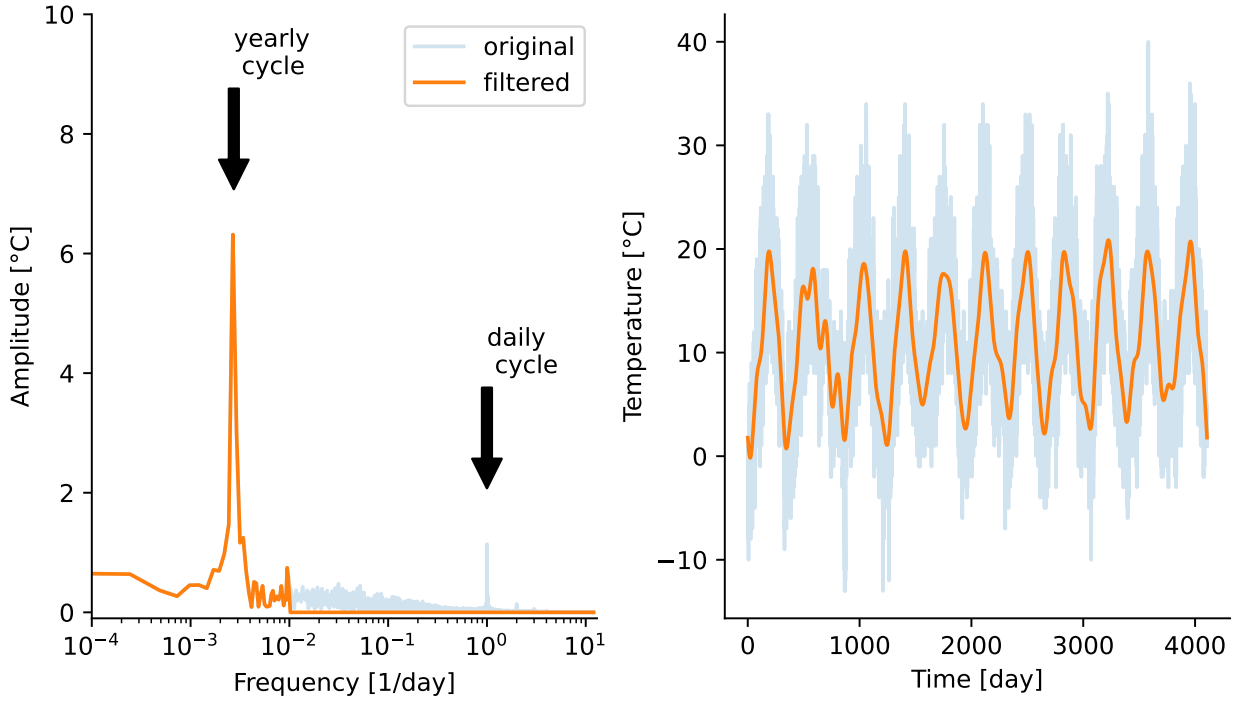
(continued from previous page)

```
    # Remove high-frequency data and transform back.
    # Suffix _f stands for "filtered".
    spectrum_f = spectrum * (freqs < 1 / 100)
    data_f = irfft(spectrum_f)

    plot_filter_results(freqs, data, spectrum, data_f, spectrum_f, "filter_
↪slow")


def plot_filter_results(freqs, data, spectrum, data_f, spectrum_f, num):
    """Plot the original and filtered results.

    Parameters
    ----------
    freqs
        The frequency axis.
    data
        The hourly temperatures.
    spectrum
        The real FFT of the hourly temperatures.
    data_f
        The filtered hourly temperatures.
    spectrum_f
        The real FFT of the filtered hourly temperatures.
    num
        A figure number, which can also be a string.

    """
    plt.close(num)
    fig, (ax0, ax1) = plt.subplots(1, 2, figsize=(7, 4), num=num)
    ax0.annotate(
        "daily\n cycle",
        xy=(1, 2),
        xytext=(1, 4),
        arrowprops=dict(facecolor="black", shrink=0.04),
    )
    ax0.annotate(
        "yearly\n cycle",
        xy=(1 / 365.0, 7),
        xytext=(1 / 365, 9),
        arrowprops=dict(facecolor="black", shrink=0.04),
    )
    norm = len(spectrum)
    ax0.plot(freqs, abs(spectrum) / norm, alpha=0.2, label="original")
    ax0.plot(freqs, abs(spectrum_f) / norm, label="filtered")
    ax0.legend()
    ax0.set_xlim(1e-4, 13)
    ax0.set_ylim(-0.1, 10)
    ax0.set_xscale("log")
    ax0.set_xlabel("Frequency [1/day]")
    ax0.set_ylabel("Amplitude [°C]")

    ax1.plot(np.arange(len(data)) / 24, data, alpha=0.2)
    ax1.plot(np.arange(len(data_f)) / 24, data_f)
```

(continues on next page)

(continued from previous page)

```
    ax1.set_xlabel("Time [day]")
    ax1.set_ylabel("Temperature [°C]")


filter_slow()
```



Due to global warming, the input data is not exactly periodic and shows a slightly increasing trend. (This is on a time span of as little as 11 years!) This would be a typical example where the discrete cosine transform would be a slightly better algorithm.

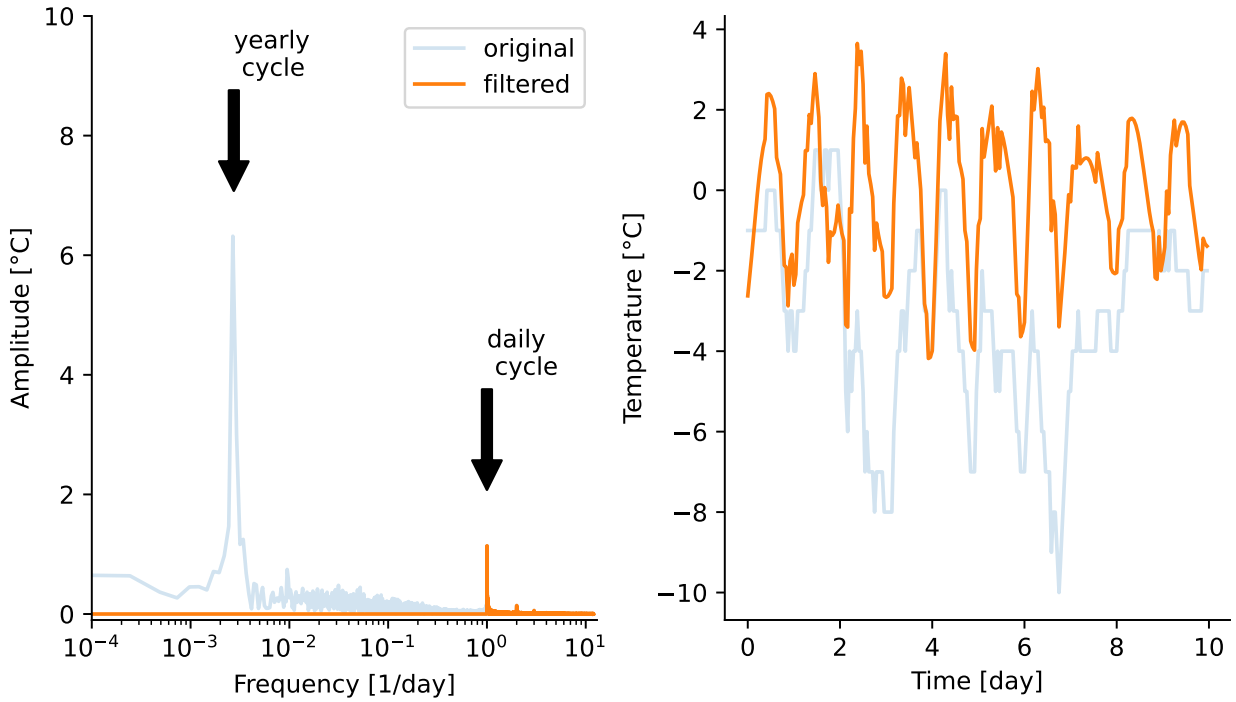
Let's now look at short-term fluctuations to get the typical Belgian daily temperature changes, without any longer-term fluctuations. For this, all fluctuations slower than 1 day will be set to zero. Only the first 10 days are plotted for the sake of clarity.

```
def filter_fast():
    """Retain only the daily fluctuations and plot the result."""
    # Load the data and compute the FFT.
    data = np.loadtxt("fft_weather.txt")
    spectrum = rfft(data)
    freqs = rfftfreq(len(data), 1 / 24)

    # Remove low-frequency data and transform back.
    # Suffix _f stands for "filtered".
    spectrum_f = spectrum * (freqs >= 1.0)
    data_f = irfft(spectrum_f)

    plot_filter_results(
        freqs,
        data[: 24 * 10],
```

(continues on next page)

(continued from previous page)

```
        spectrum,
        data_f[: 24 * 10],
        spectrum_f,
        "filter_fast",
    )


filter_fast()
```



11.4.2 Efficient calculation of convolutions

The discrete circular convolution of two **periodic** sequences $\mathbf{u}$ and $\mathbf{v}$ of length n is defined as

$$\{\mathbf{u} * \mathbf{v}\}_m = \sum_{k=0}^{n-1} v_k u_{m-k}, \quad \forall\, m \in \{0, 1, \dots, n-1\}.$$

The periodicity means that $v_k = v_{k+n}$ and $u_k = u_{k+n}$. Still, the arrays in computations only contain values for only one period.

This operation is equivalent to multiplication by a **circulant matrix**

$$\begin{bmatrix} z_0 \\ z_1 \\ \vdots \\ z_{n-2} \\ z_{n-1} \end{bmatrix} = \begin{bmatrix} u_0 & u_{n-1} & u_{n-2} & \cdots & u_1 \\ u_1 & u_0 & u_{n-1} & \cdots & u_2 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ u_{n-2} & \cdots & u_1 & u_0 & u_{n-1} \\ u_{n-1} & \cdots & u_2 & u_1 & u_0 \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-2} \\ v_{n-1} \end{bmatrix}$$

Such a matrix is diagonalized by the DFT, thus:

$$\begin{bmatrix} \hat{z}_0 \\ \hat{z}_1 \\ \vdots \\ \hat{z}_{n-2} \\ \hat{z}_{n-1} \end{bmatrix} = \begin{bmatrix} \hat{u}_0 & 0 & \cdots & \cdots & 0 \\ 0 & \hat{u}_1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & \hat{u}_{n-2} & 0 \\ 0 & \cdots & \cdots & 0 & \hat{u}_{n-1} \end{bmatrix} \begin{bmatrix} \hat{v}_0 \\ \hat{v}_1 \\ \vdots \\ \hat{v}_{n-2} \\ \hat{v}_{n-1} \end{bmatrix}$$

For this reason, it is more efficient to use the FFT algorithm to transform the inputs to the frequency domain, compute one pointwise multiplication, and transform the result back to the time domain.

> **Example**
>
> Let's calculate the convolution of the two sequences $\mathbf{u} = [0110]$ and $\mathbf{v} = [1100]$
>
> $$\begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 1 \end{bmatrix}$$
>
> Let's now verify this answer by using python to calculate
>
> $$\mathbf{z} = \mathscr{F}^{-1}\left(\mathscr{F}(\mathbf{u})\mathscr{F}(\mathbf{v})\right)$$
>
> where $\mathscr{F}$ and $\mathscr{F}^{-1}$ denote the FFT and its inverse, respectively.

```
def demo_convolve_rfft():
    """Show how to compute a convolution with RFFT."""
    a = np.array([0, 1, 1, 0])
    b = np.array([1, 1, 0, 0])
    ffta = rfft(a)
    fftb = rfft(b)
    fftz = ffta * fftb
    # Note: the length is added as argument.
    # If not, it will always return a sequence of even length.
    z = irfft(fftz, len(a))
    # Note: suppress_small=True hides the rounding errors.
    print(np.array2string(z, suppress_small=True))


demo_convolve_rfft()
```

```
[0. 1. 2. 1.]
```

The same result could also be obtained with `scipy.signal.convolve`. We use the `method="direct"` option to make sure it does not use the FFTs for the calculation. (By default, it will take the fastest method for the given lengths of the inputs.)

```
def demo_convolve_scipy():
    """Compute the convolution with SciPy's convolve function."""
    a = np.array([0, 1, 1, 0])
    b = np.array([1, 1, 0, 0])
    z = signal.convolve(a, b, method="direct")
    print(z)


demo_convolve_scipy()
```

```
[0 1 2 1 0 0 0]
```

The additional zero padding returned by `scipy.signal.convolve` becomes intuitively clear by viewing the convolution as the sum of the overlap, as a function of the shift, as the two sequences are shifted over each other.

0	1	1	1 0	1	0	0	pointwise multiplication
0	0	0	0	0	0	0	sum = 0

0	1	1 1	1 0	0	0	
0	0	1	0	0	0	sum = 1

0	1 1	1 1	0 0	0	
0	1	1	0	0	sum = 2

1 0	1 1	0 1	0 0	
0	1	0	0	sum = 1

1	1 0	0 1	0 1	0	
0	0	0	0	0	sum = 0

1	1	0 0	0 1	1	0	
0	0	0	0	0	0	sum = 0

1	1	0	0 0	1	1	0	
0	0	0	0	0	0	0	sum = 0

The zero-padding can also be achieved with the FFT algorithm, by appropriately zero-padding the inputs.

```
def demo_convolve_rfft_padding1():
    """Show how to compute a convolution with RFFT."""
    a = np.array([0, 1, 1, 0, 0, 0, 0])
    b = np.array([1, 1, 0, 0, 0, 0, 0])
    ffta = rfft(a)
    fftb = rfft(b)
    fftz = ffta * fftb
    z = irfft(fftz, len(a))
    print(np.array2string(z, suppress_small=True))


demo_convolve_rfft_padding1()
```

```
[-0.  1.  2.  1.  0.  0.  0.]
```

Zero-padding is also the correct method to compute convolutions of non-periodic inputs, assuming the data should be zero outside the range where it is defined. Because this is such a common operation, the `fft` and `rfft` functions have an optional argument `n` to specify the desired zero-padding of the input.

```
def demo_convolve_rfft_padding2():
    """Show how to compute a convolution with RFFT."""
    n = 7
    a = np.array([0, 1, 1, 0])
    b = np.array([1, 1, 0, 0])
```

(continues on next page)

(continued from previous page)

```
    ffta = rfft(a, n)
    fftb = rfft(b, n)
    fftz = ffta * fftb
    z = irfft(fftz, n)
    print(np.array2string(z, suppress_small=True))


demo_convolve_rfft_padding2()
```

```
[-0.  1.  2.  1.  0.  0.  0.]
```

11.4.3 Autocorrelation

The **autocorrelation** of a real sequence $\mathbf{y}$ expresses the similarity between a sequence and a delayed copy of itself. It is defined as

$$\mathbf{R}_\ell = \sum_{k=0}^{n-1} \mathbf{y}_k \mathbf{y}_{k-\ell} \quad \forall\, l \in \{0, 1, \ldots, n-1\},$$

We recognize that this is a convolution of the sequence with a **reversed copy** of itself, i.e. $\mathbf{y}_{k-\ell}$ instead of $\mathbf{y}_{\ell-k}$, and thus can be calculated efficiently using FFTs as

$$\mathbf{R} = \mathscr{F}^{-1}\left(\mathscr{F}(\mathbf{y})(\mathscr{F}(\mathbf{y})^*)\right) = \mathscr{F}^{-1}\left(\|\mathscr{F}(\mathbf{y})\|^2\right)$$

where * denotes a complex conjugate. The complex conjugation is due to the reversal of the second convolution argument in the time domain.

If we want to get the **autocovariance** instead of the **autocorrelation**, we need to *demean* (i.e. subtracting the mean of the sequence) the sequence:

$$\mathbf{K} = \mathscr{F}^{-1}\left(\|\mathscr{F}(\mathbf{y} - \overline{\mathbf{y}})\|^2\right)$$

It is also common to further divide by the *variance* of the sequence, which is conveniently calculated from the first entry in the Fourier transformed sequence.

$$\rho = \frac{\mathbf{K}}{\sigma_y^2}$$

(There is no proper name for the latter quantity. It is often called autocorrelation or autocovariance, while it is actually neither.)

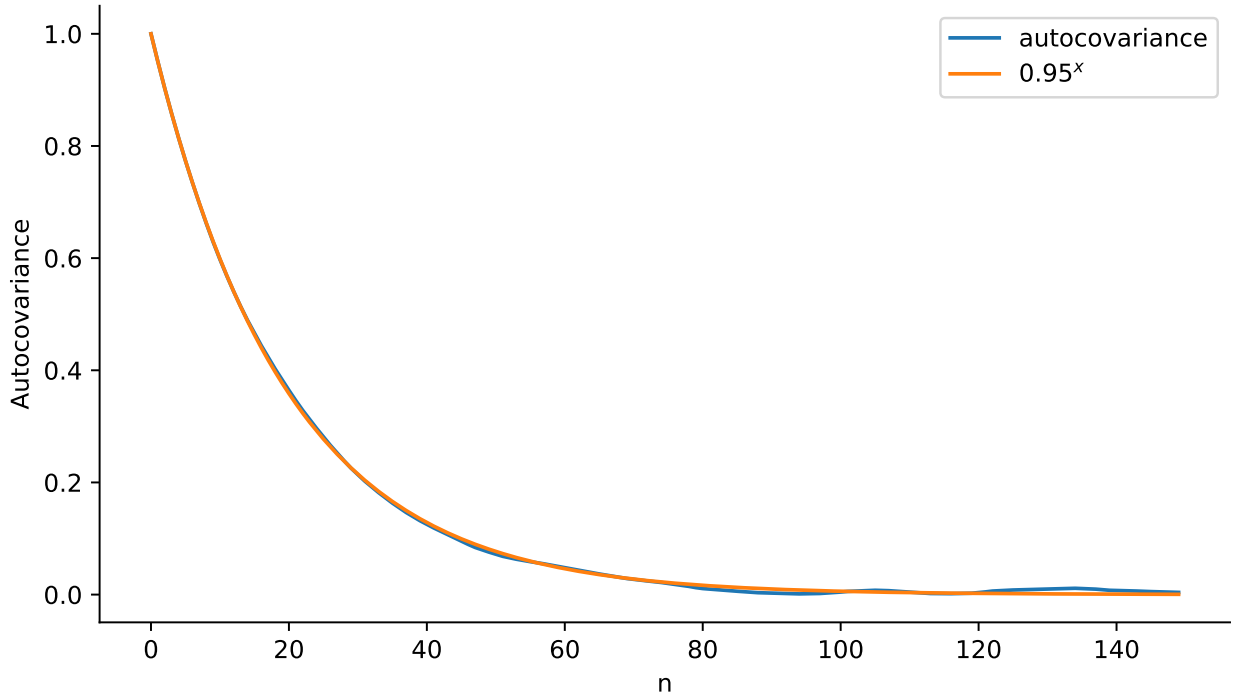
> **Example**
>
> As an example, we'll generate a random sequence where each number is determined by taking 95% of the previous number and adding 5% of a random number to this value. Note that we generate our random values between -1 and 1, so that the mean will be 0.
>
> We thus expect that the correlation between each value and a value n positions further scales as 0.95^n.

```
def demo_autocovariance_fft():
    """Demonstrate the computation of an autocovariance with RFFT."""
    # Generate the stochastic input.
    # The calculation is unavoidably sequential,
    # so no vectorization possible.
```

(continues on next page)

(continued from previous page)

```
    rng = np.random.default_rng()
    a = np.zeros(100000)
    a[0] = rng.uniform(-1, 1)
    for i in range(1, len(a)):
        a[i] = a[i - 1] * 0.95 + 0.05 * rng.uniform(-1, 1)

    # Compute auto-correlation.
    # In principle, we would have to use padding,
    # but it has no noticeable effect in practice.
    y = rfft(a)
    R = irfft(abs(y) ** 2)

    # Compare to the analytical result in a plot.
    x = np.arange(150)
    f = (0.95) ** x

    plt.close("autocovariance")
    fig, ax = plt.subplots(figsize=(7, 4), num="autocovariance")
    ax.plot(x, R[: len(x)] / R[0], "-", label="autocovariance")
    ax.plot(x, f, "-", label="$0.95^x$")
    ax.set_xlabel("n")
    ax.set_ylabel("Autocovariance")
    plt.legend()


demo_autocovariance_fft()
```



There are many more physical quantities which used to be very computationally demanding to compute, but which can luckily be computed (a bit) more efficiently using the Fast Fourier Transform algorithm (although typically, these computations remain the bottleneck in simulation programs).

One example is the long-range magnetostatic interaction, which in a naive implementation requires $\mathcal{O}(n^2)$ calculations.

(n interactions for each of the n spins in the system). By recasting the equations in a suitable way, this can be written as a convolution, which allows for a more efficient computation.

11.4.4 Fast polynomial multiplication

Another example of a calculation that can be sped up using FFT's is the multiplication of two polynomial functions.

To find the coefficients of the product of two polynomials $f(x) = f_1(x) \cdot f_2(x)$, we need to:

- calculate all the pair-wise products of their respective terms,
- group these products by the order of the resulting mononomial,
- and sum the products within each group.

When we write the coefficients as vectors $\mathbf{f}_1$ and $\mathbf{f}_2$ (ordered from the lowest order to the highest; i.e. the constant term first), and append zeros such that the vectors have a dimension larger than the degree of f_1 + the degree of f_2, we can write the polynomial product as a convolution, where the coefficients of their product $\mathbf{f}$ are given by:

$$\mathbf{f} = \mathbf{f}_1 * \mathbf{f}_2$$

We saw earlier that we can calculate this in an efficient way using Fast Fourier Transforms:

$$\mathbf{f} = \mathscr{F}^{-1}\big(\mathscr{F}(\mathbf{f}_1)\mathscr{F}(\mathbf{f}_2)\big)$$

Example

Let's try to find the product of the functions

$$f_1 = 3 + 2x + x^2 + x^3$$

and

$$f_2 = 1 + x + x^2 + x^3.$$

The corresponding coefficient vectors are

$$\mathbf{f}_1 = [3, 2, 1, 1, 0, 0, 0] \quad \text{and} \quad \mathbf{f}_2 = [1, 1, 1, 1, 0, 0, 0].$$

Note the zero padding to the size needed to represent the product (up to x^6).

As shown below, we find that the product of f_1 and f_2 equals

$$\begin{aligned}\mathbf{f} &= \mathscr{F}^{-1}\big(\mathscr{F}(\mathbf{f}_1)\mathscr{F}(\mathbf{f}_2)\big) \\ &= [3, 5, 6, 7, 4, 2, 1]\end{aligned}$$

where $\mathscr{F}$ and $\mathscr{F}^{-1}$ denote the FFT and its inverse, respectively. Thus:

$$f(x) = f_1(x) \cdot f_2(x) = 3 + 5x + 6x^2 + 7x^3 + 4x^4 + 2x^5 + 1x^6$$

```
def multiply_polys(a1, a2):
    """Multiply two polynomials, whose coefficients are given in a1 and a1."""
    # Zero padding is used to make sure the output is
    # sufficiently large for the term with the highest degree.
    length = len(a1) + len(a2) - 1
    y1 = rfft(a1, length)
```

(continues on next page)

(continued from previous page)

```
    y2 = rfft(a2, length)
    return irfft(y1 * y2, length)


def eval_poly(a, x):
    """Evaluate a polynomial in with coefficients a in point x."""
    # The following is the product of a column
    # from a Vandermonde matrix and a vector.
    return np.dot(x ** np.arange(len(a)), a)


def demo_poly():
    """Demonstrate polynomial multiplication through RFFT."""
    # you can check for yourself that for any x, you'll get the correct result
    x = 3.0
    a1 = np.array([3, 2, 1, 1])
    a2 = np.array([1, 1, 1, 1])
    a3 = multiply_polys(a1, a2)
    print("Coefficients of polynomial product:", a3)
    print("Evaluate product of polynomials:", eval_poly(a1, x) * eval_poly(a2,
 ↪ x))
    print("Evaluate polynomial product:", np.abs(eval_poly(a3, x)))


demo_poly()
```

```
Coefficients of polynomial product: [3. 5. 6. 7. 4. 2. 1.]
Evaluate product of polynomials: 1800.0
Evaluate polynomial product: 1799.9999999999995
```

11.5 SciPy resources

All details of the `fft`, `rfft` and related functions we used throughout this notebook can be found on

https://docs.scipy.org/doc/scipy/reference/fft.html

Furthermore, an instructive tutorial in the `scipy` user guide is available on

https://docs.scipy.org/doc/scipy/tutorial/fft.html

CHAPTER 12

Monte Carlo (MC)

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.patches import Rectangle
from scipy import stats
from scipy.integrate import quad, solve_ivp
```

Broadly speaking, a Monte Carlo method is a **numerical technique that employs (pseudo) random numbers**. Despite the randomness, results obtained with a Monte Carlo method are **intended to be reproducible**. This seems counter-productive at first sight: why should one make use of random numbers, if the result of interest is not random? As we'll see in this chapter, Monte Carlo methods enable computations that would be intractable otherwise.

In general, Monte Carlo belongs in a statistics course, yet **many physical systems show stochastic behavior**, such that Monte Carlo methods are a good fit to model them. The stochastic behavior can be due to the chaotic behavior of their dynamical equations (e.g. three-body problem) or simply because the governing laws of physics are inherently stochastic (e.g. radioactive decay). In such cases, the physics of interest is best described by a probability distribution. Properties of interest are then computed as expectation values of such a distribution.

When a Monte Carlo method is used in a computational model of a physical system, one often uses the term *Monte Carlo simulation*. The name *Monte Carlo* was introduced by John von Neumann and Stanley Ulam as a code word in the Manhattan Project, for simulations used to design the first nuclear weapons. Their methodology became so popular, in all scientific disciplines, such that the code word quickly replaced more scientific terms such as "Model Sampling".

This chapter presents merely a first introduction to the Monte Carlo method. Before explaining Monte Carlo, pseudo-random numbers are briefly reviewed. This is followed by two more sections. First, the "basic Monte Carlo" method is explained, in which case independent random numbers can be easily generated with library routines. Finally, "Markov chain Monte Carlo" methods are introduced, in which case a statistical process is implemented to sample a nontrivial distribution. In all sections, we will mainly focus on continuous probability densities, yet the same techniques can also be applied to discrete distributions.

12.1 Pseudo-random number generator (PRNG)

Computer hardware is designed to carry out purely deterministic arithmetic operations. **Any computational result is therefore not truly random.** Nonetheless, one may design algorithms that produce a sequence of seemingly uncorrelated numbers.

The simplest algorithm in this category is the *Linear Congruential Generator* (LCG). Note that simplicity is its only advantage: for most applications, the correlation between subsequent values is too obvious, resulting in biased outcomes of Monte Carlo simulation. It is only used here to illustrate the basic structure of any PRNG. **Never use LCG for serious applications, and keep in mind that many software libraries still implement it.**

The following code cell implements LCG.

```
def generate_lcg(size, seed, factor, offset, divisor):
    """Generate a LCG sequence.

    Parameters
    ----------
    size
        The number of generated pseudo-random numbers,
        also the size of the returned 1D array.
    seed
        The initial value.
    factor, offset, divisor
        Parameters in the LCG algorithm.

    Returns
    -------
    sequence
        A 1D array containing the LCG sequence.

    """
    result = np.zeros(size, dtype=int)
    state = seed
    result[0] = state
    for i in range(1, size):
        state = (factor * state + offset) % divisor
        result[i] = state
    return result


generate_lcg(24, 3, factor=7, offset=1, divisor=13)
```

```
array([ 3,  9, 12,  7, 11,  0,  1,  8,  5, 10,  6,  4,  3,  9, 12,  7, 11,
        0,  1,  8,  5, 10,  6,  4])
```

12.1.1 Key ingredients of a PRNG

The LCG algorithm contains the four **key ingredients** also found in other PRNGs:

1. A `seed` must be provided, which determines the entire random sequence. Some implementations use the current time as the seed when no seed is provided by the user. For example, one may use the number of seconds since the epoch (January 1st, 1970).
2. The algorithm has an internal `state`, which is initialized with the seed and is updated after a new number is generated. Advanced PRNGs conceptually extend the state in several ways:
   - In general, the state can be an array of numbers, or a binary (0 or 1) array of some size.
   - When the seed contains too few bits of information, it can be padded with other values to fill up the initial state.
   - The random number generated at each sequence may be a function of the state, rather than being equal to the state in the simple case of LCG.

3. A **recurrence relation** is used to derive the next state from the current one.
4. Fixed **parameters** appearing in the algorithm.

12.1.2 Properties of pseudo-random sequences

All pseudo-random sequences have the following **properties**, which can be recognized in the output of the LCG example above:

1. The sequence is **deterministic**: repeating the calculation with the same seed and parameters gives the same result.
2. There is only a **limited number of pseudo-random values**. The upper limit is given by 2^N, where N is the number of bits used to represent the state. In practice, algorithms have fewer different random numbers:
   - The algorithm may limit the range of the pseudo-random numbers by construction.

     > In the LCG example, the modulo operator limits the values to the set $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$.

   - Less obvious, the recurrence relation may not be capable of visiting all potential values.

     > In the LCG example, 2 is not present in the sequence. (Try 2 as a seed.)

3. Due to the previous points, **pseudo-random sequences must be periodic** (possibly after an initialization phase). When the same state is encountered as before, all subsequent states will be repeated as well. Because the number of states is limited, it is unavoidable that, after some time, the same state appears again. In the above LCG example, the period is 12, not 13.

   In practice, PRNGs are designed with well-defined periods and without any initialization phase.

12.1.3 Desirable characteristics of a PRNG

To avoid bias in simulations with pseudo-random sequences, a good PRNG algorithm should have the following two main characteristics:

1. The pseudo-random numbers should be **uniformly distributed**.
   - Ideally, all possible states in a sequence can be visited (from any seed). This would imply that each state appears with the same probability, resulting in a uniform distribution.
   - Several isolated periodic pseudo-random subsequences may exist in the space of all states. In the LCG example above, the subsequences are $\{2\}$ and $\{3, 9, 12, 7, 11, 0, 1, 8, 5, 10, 6, 4\}$. When using a seed different from 2, the distribution is not uniform and has a gap at 2.
2. There should be **no apparent statistical correlations** between subsequent values.
   - A minimal requirement is that pseudo-random sequences have **long periods**, ideally much longer than the amount of random numbers needed in any application.
   - Also, after **differentiating** the sequence, no obvious correlation should appear. For example, the spacing between two subsequent random numbers should also be pseudo-random.
   - In addition, when generating N-dimensional **vectors of random numbers**, these should be distributed uniformly throughout their space.

The desirable characteristics can to some extent be assessed by a pen-and-paper analysis, which is useful for designing new algorithms. Moreover, standard numerical tests exist to validate the desirable characteristics of a PRNG, most notably TestU01. For an effective algorithm, both the operations and the parameters need to be carefully selected. NumPy uses the PCG64 algorithm by default, which performs well in both categories and has several additional technical advantages.

12.1.4 Modern PRNG algorithms

A complete exposition of the history of PRNG development goes beyond the scope of this course. However, it is worth looking at one popular family of modern PRNGs, namely the Xorshift methods by Marsaglia.

Modern algorithms often make use of binary operators such as `xor` and `shift`, because they are computationally efficient:

- In Python, `or` is implemented with the operator `|`. It applies the *bitwise or* to a pair of integers. For every corresponding pair of bits, `or` is computed as follows:

In1	In2	Out
0	0	0
0	1	1
1	0	1
1	1	1

  When applied to integer numbers, `|` has the following effect.

Python	Decimal	Binary
`a`	5	0101
`b`	12	1100
`a | b`	13	1101

- `xor` is implemented with the operator `^`. It applies the *bitwise exclusive or* to a pair of integers. For every corresponding pair of bits, `xor` is computed as follows:

In1	In2	Out
0	0	0
0	1	1
1	0	1
1	1	0

  When applied to integer numbers, `^` has the following effect.

Python	Decimal	Binary
`a`	5	0101
`b`	12	1100
`a ^ b`	9	1001

- `shift` shifts all the bits in the binary representation of an integer to the left or the right. Left and right shifts are implemented in Python with the `<<` and `>>` operators, respectively. For example:

Python	Decimal	Binary
	5	00101
`5 << 1`	10	01010
`5 << 2`	20	10100
`5 >> 1`	2	00010

Shifting to the left multiplies by 2, while shifting to the right is a division by 2. Whenever bits are shifted out of the register, they are discarded.

- `bitroll` is not a low-level operation (and has no official name either), but is popular in modern PRNGs. It combines two shift operators to permute bits in a binary number. The following table contains some examples for 4-bit integers:

Input	Decimal	roll	Output	Decimal
0010	2	1	0100	4
0101	5	1	1010	10
1001	9	1	0011	3
0011	3	2	1100	12
0011	3	3	1001	9

```
def bitroll64(x, shift):
    """Apply a bitwise roll operation on 64-bit integers.

    Parameters
    ----------
    x
        The number to be bitrolled.
    shift
        The number of bits to be shifted to the left.

    Returns
    -------
    xroll
        The bits of x are shifted ``shift`` to the left.
        Any bits exceeding the 64-bit register are inserted
        again on the right.

    """
    return (np.uint64(x) << np.uint64(shift)) | (
        np.uint64(x) >> np.uint64(64 - shift)
    )


assert bitroll64(0, 45) == 0
assert bitroll64(1, 1) == 2
assert bitroll64(3, 2) == 12
assert bitroll64(3, 63) == 1 + 2**63
```

The following code cell implements the `xoshiro256**` variant in the Xorshift family, proposed in 2018, and gives you a reasonable idea of a state-of-the-art algorithm. The state consists of four unsigned 64-bit integers.

The code below was derived from a C implementation.

Note that the C source code is much simpler, because it relies on something that is typically avoided in Python: when the product of two integers does not fit in the 64-bit register, the most significant bits are just discarded, a counter-intuitive behavior exploited by PRNGs. Because this typical C behavior easily leads to bugs, Python and NumPy complain when this happens, and some extra effort is needed to suppress such warnings.

```
def generate_xoshiro256ss(size, seed, pars=None):
    """Generate a xoshiro256** sequence of unsigned 64-bit integers.
```

(continues on next page)

(continued from previous page)

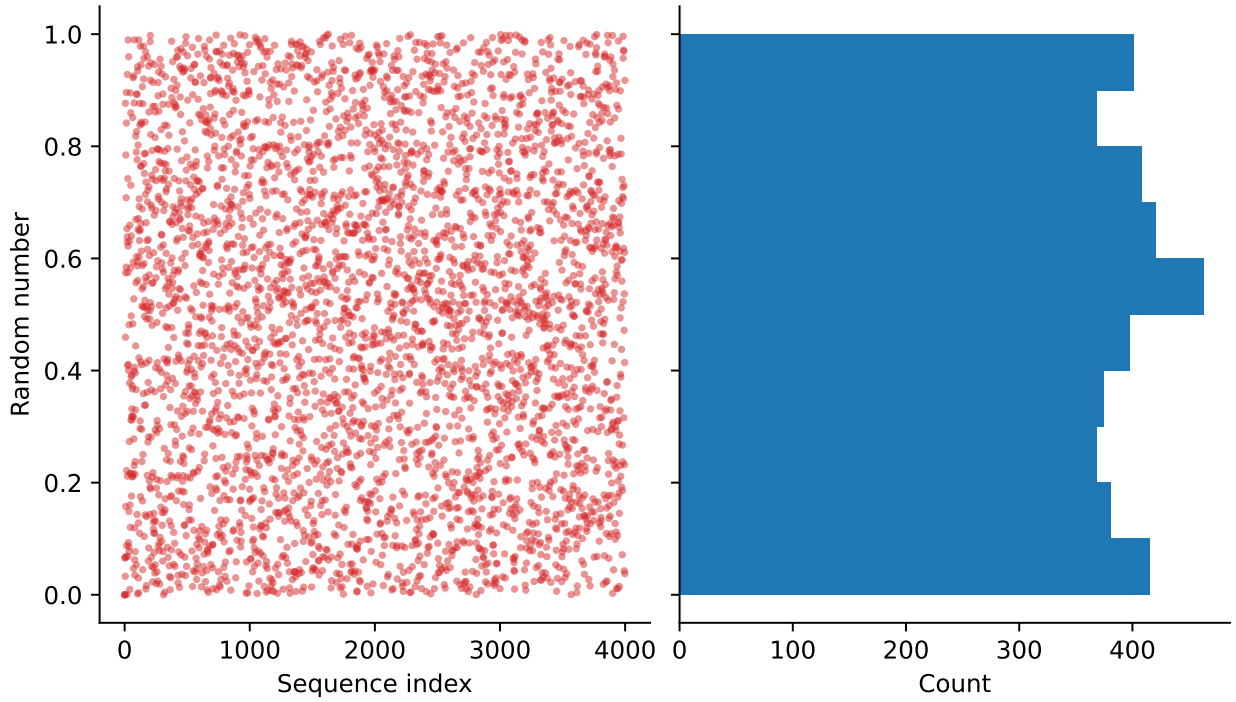
```
    Parameters
    ----------
    size
        The amount of numbers to generate.
    seed
        An initial state for the generator.
        This must be a 1D array with exactly 4 unsigned
        64-bit integers.
    pars
        The five integer parameters for the algorithm.
        The default is [5, 7, 9, 17, 45]

    Returns
    -------
    sequence
        A 1D array containing the pseudo-random sequence.

    """
    state = np.array(seed, dtype=np.uint64)
    if state.shape != (4,):
        raise TypeError("The seed must be a sequence of 4 integers.")

    if pars is None:
        pars = np.array([5, 7, 9, 17, 45], np.uint64)
    else:
        pars = np.asarray(pars, dtype=np.uint64)
        if pars.shape != (5,):
            raise TypeError("The parameters must be a sequence of 5 integers.
 ↪")

    result = np.zeros(size, dtype=np.uint64)
    for i in range(size):
        result[i] = np.multiply(
            bitroll64(np.multiply(state[1], pars[0], dtype=np.uint64),␣
 ↪pars[1]),
            pars[2],
            dtype=np.uint64,
        )
        t = state[1] << pars[3]
        state[2] ^= state[0]
        state[3] ^= state[1]
        state[1] ^= state[2]
        state[0] ^= state[3]
        state[2] ^= t
        state[3] = bitroll64(state[3], pars[4])
    return result


assert (
    generate_xoshiro256ss(4, [1, 2, 3, 4])
    == [11520, 0, 1509978240, 1215971899390074240]
).all()
print(generate_xoshiro256ss(9, [1, 2, 3, 4]))
```

```
[                11520                        0           1509978240
  1215971899390074240  1216172134540287360   607988272756665600
 16172922978634559625  8476171486693032832 10595114339597558777]
```

The first three values are noticeably low due to the choice of the seed. Afterward, the distribution is uniform over the whole range of unsigned 64-bit integers. For this algorithm, it is recommended to use another PRNG (`splitmix64`) to initialize the seed.

One can easily convert 64-bit integers to uniformly distributed random numbers with double precision on the interval [0, 1] by taking the leading 53 bits of each number and multiplying them with `2e-53`.

```
def plot_xoshiro256ss():
    """Plot the sequence of random numbers generated by xoshiro256ss."""
    plt.close("xoshiro256ss")
    fig, axs = plt.subplots(1, 2, figsize=(7, 4), sharey=True, num=
 ↪"xoshiro256ss")
    seq_int64 = generate_xoshiro256ss(4000, [1, 2, 3, 4])
    seq_flt64 = (seq_int64 >> 11) * np.finfo(np.float64).epsneg
    axs[0].plot(seq_flt64, ".", color="C3", alpha=0.5, mew=0)
    axs[0].set_ylabel("Random number")
    axs[0].set_xlabel("Sequence index")
    axs[1].hist(seq_flt64, bins=10, orientation="horizontal")
    axs[1].set_xlabel("Count")


plot_xoshiro256ss()
```



For Monte Carlo, **Xorshift methods (and many others) are practically sufficient and have a low computational cost**. An older popular method is the Mersenne-Twister algorithm, proposed in 1997, but both its computational performance and quality of randomness are outperformed by more recent algorithms. Ongoing research on PRNG algorithms is trying to establish a better trade-off between the desirable characteristics and computational cost.

Because the **recurrence relations are inherently serial**, one cannot simply generate random numbers in parallel.

Hence, for the sake of computational efficiency, PRNGs are typically implemented in low-level code (not Python). Vectorization and parallelism are sometimes used to produce multiple streams of parallel random numbers for high-performance applications.

In addition to Monte Carlo, another major application of random numbers is **cryptography**. This application comes with additional algorithm requirements, generally related to the predictability of pseudo-random sequences.

- Given an example sequence, one may easily detect the algorithm that was used and its internal state. Once these are determined, an adversary knows your future random numbers and can guess your random encryption keys. This can be prevented by applying a non-invertible hash function to random data.
- In extreme cases, predictability can be further reduced by including stochastic seeds from physical processes, such as radio-active decay, thermal noise, lava lamps, etc.

12.1.5 Transformations of univariate continuous distributions

Uniformly distributed numbers can be transformed, to sample other continuous univariate distributions. Two common methods are mentioned here for the sake of completeness:

- Inverse transform sampling. Given a random variable X uniformly distributed over $[0, 1]$, it can be transformed to $Y = F_Y^{-1}(X)$, where F_y is the cumulative distribution of the quantity Y.
- The Box-Muller transform is an efficient method for sampling a standard normal distribution.

The details of both methods are skipped here, because the Monte Carlo examples below rely on functionality built into NumPy and SciPy to generate random numbers for various distributions.

12.1.6 Built-in, NumPy and SciPy implementations

Built-in `random` (which should not be used)

The built-in Python package `random` uses the good old Mersenne Twister algorithm. While this is proven technology, the built-in package has some limitations that make it unappealing for computational use:

- More efficient PRNGs have been developed.
- The built-in package cannot efficiently create arrays of random numbers.
- The number of implemented statistical distributions is limited.

As a rule of thumb, never use the built-in `random` package for scientific purposes.

NumPy

The default random number generator in NumPy is the 64-bit Permuted Congruential Generator, which is another modern PRNG. NumPy implements a reasonable set of statistical distributions, see numpy.random.Generator.

Example showing how to fill a 5×3 array by uniformly sampling over $[10^{-3}, 10^3]$:

```
np.random.uniform(-1e3, 1e3, (5, 3))
```

```
array([[-683.76388984,  146.00286442, -712.93568643],
       [-646.68196491,  901.39647713,  580.35777695],
       [-620.83249566, -470.13305857,  891.17530784],
       [-627.46919819, -559.40652046, -606.18074051],
       [ 424.31355035,  471.3405622 ,  892.00886927]])
```

The example above uses the outdated function-based interface to `numpy.random`, which is no longer recommended as of NumPy 1.17.0 (July 2019).

It is recommended to use the new object-oriented interface: you first create an `rng` object with its own internal state and methods of the `rng` object can be called to get the actual random numbers.

The advantage of the `rng` object is that you have full control over changes to the internal state of the RNG algorithm, which is not the case with the old function-based API. If your program and another library both use the same RNG state, you may get unpredictable random numbers, even with a fixed seed. This can be very confusing when trying to debug a program using random numbers.

For example, you can create an `rng` object with a fixed seed as follows:

```
def demo_numpy_rng():
    rng = np.random.default_rng(1)
    print(rng.uniform(-1e3, 1e3, (5, 3)))


demo_numpy_rng()
```

```
[[  23.6432494   900.92739265 -711.68077456]
 [ 897.29889427 -376.33709598 -153.34710205]
 [ 655.40518764 -181.60172726   99.18737535]
 [-944.88177351  507.02621735   76.28662644]
 [-340.536567    576.85740686 -393.61034142]]
```

SciPy

SciPy supports a much broader selection of probability densities through the `scipy.stats` module.

Example showing how to fill a 5×3 array by sampling a continuous Chi-squared distribution with 8 degrees of freedom:

```
def demo_scipy_rng():
    rng = np.random.default_rng(1)
    print(stats.chi2.rvs(8, size=(5, 3), random_state=rng))


demo_scipy_rng()
```

```
[[ 8.73803557  8.67299915 11.37573554]
 [ 5.46318272  8.82002249  7.44272167]
 [ 4.85905393  5.63758124  7.48651174]
 [ 4.72794652  7.36455973 13.48945084]
 [ 1.07974111  6.68416419  8.18232983]]
```

The `scipy.stats` module can also compute various other properties of distributions, such as the PDF, CDF, etc. See, for example, the examples for the Chi-squared distribution.

12.1.7 References

Mersenne Twister

- Matsumoto, M.; Nishimura, T. (1998). "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator". ACM Trans. Mod. Comput. Sim. **8** (1), pp 3–30. 10.1145/272991.272995.

Xorshift

- Marsaglia, G. (2003). "Xorshift RNGs". *J. Stat. Soft.*, **8** (14), 1–6. 10.18637/jss.v008.i14

Xoshira256**

- Blackman, D.; Vigna, S. (2018). "Scrambled Linear Pseudorandom Generators". arXiv:1805.01407
- Blackman, D.; Vigna, S. (2021). "Scrambled Linear Pseudorandom Generators". ACM Trans. Math. Soft. **47** (4) article no. 36, pp 1–32 10.1145/3460772

PCG64 (Used by NumPy)

- https://www.pcg-random.org/
- O'Neill, M.E. (2014). "PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation". https://www.cs.hmc.edu/tr/hmc-cs-2014-0905.pdf

12.2 Basics of the Monte Carlo method

The Monte Carlo method relies on the following identity from statistics:

$$\mathrm{E}[g(\mathbf{X})] = \int g(\mathbf{x}) p_{\mathbf{X}}(\mathbf{x})\,\mathrm{d}\mathbf{x}$$

where:

- $\mathbf{X}$ is a stochastic vector in $\mathbb{R}^n$.
- $\mathbf{x}$ is an (ordinary) vector in $\mathbb{R}^n$.
- $p_{\mathbf{X}}(\mathbf{x})$ is the probability density.
- $p_{\mathbf{X}}(\mathbf{x})\,\mathrm{d}\mathbf{x}$ is the probability of finding a sample point $\mathbf{X}$ in a region of size $\mathrm{d}\mathbf{x}$ around $\mathbf{x}$.
- $g(\cdot)$ can be any function $\mathbf{x} \mapsto g(\mathbf{x}) \in \mathbb{R}^m$.
- The integral is taken over the entire domain, where $p_{\mathbf{X}}(\mathbf{x})$ is non-zero.
- $\mathrm{E}[\cdot]$ stands for "the expectation value", assuming $\mathbf{X}$ is distributed according to the probability density $p_{\mathbf{X}}(\mathbf{x})$.

For many applications, the function g is scalar. For several examples below, also $\mathbf{X}$ and $\mathbf{x}$ are also scalar quantities.

With the above identity, one may approximate the integral in the right-hand side, just by taking N sample points $\mathbf{X}_i$ from the distribution $p_{\mathbf{X}}$, computing all $g(\mathbf{X}_i)$ and averaging over all results.

$$\int g(\mathbf{x}) p_{\mathbf{X}}(\mathbf{x})\,\mathrm{d}\mathbf{x} \approx \frac{1}{N}\sum_{i=1}^{N} g(\mathbf{X}_i) = \overline{g(\mathbf{X}_i)}$$

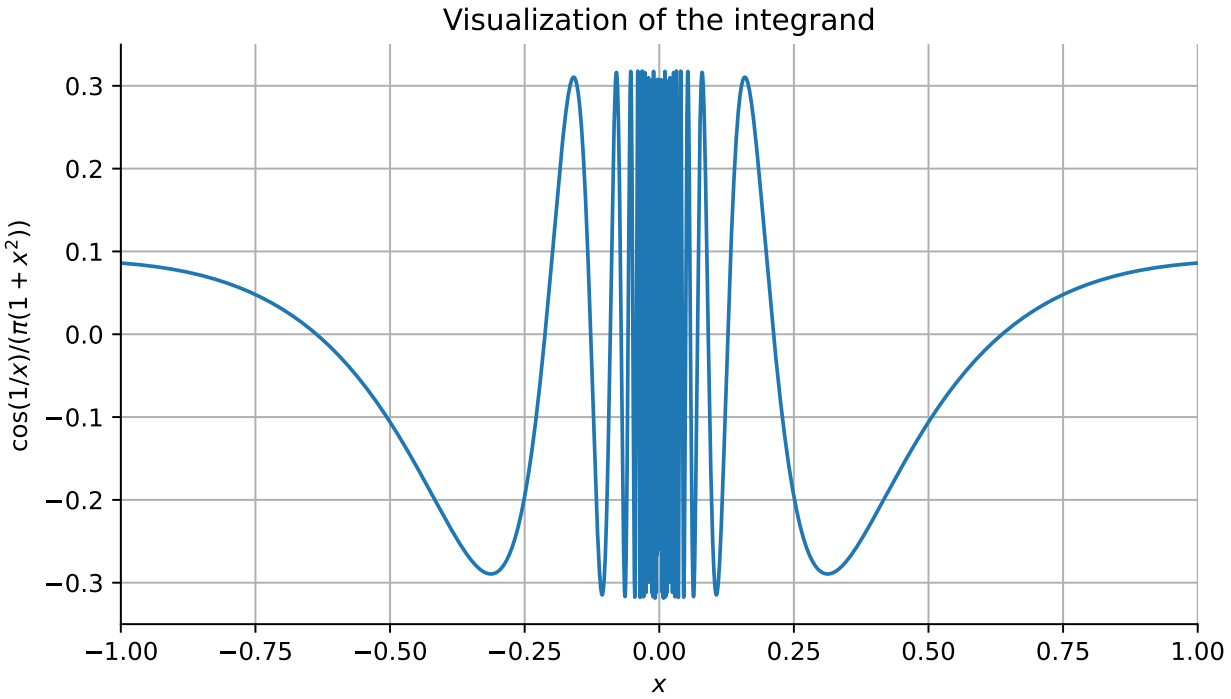
> **Example: a one-dimensional integral**
>
> Let's compute the following integral numerically, for which the analytical solution is known.
>
> $$\int_{-\infty}^{+\infty} \cos(1/x)\frac{1}{\pi(1+x^2)}\,\mathrm{d}x = \frac{1}{e} \approx 0.367879441171442$$
>
> This integral is challenging for quadrature methods due to its highly oscillatory nature.
>
> The second factor is a standard Cauchy distribution. Therefore, the integral can be approximated by $\overline{\cos(1/X_i)}$ where X_i are standard-Cauchy-distributed numbers.

```
def integrand_1d(x):
    return np.cos(1 / x) / np.pi / (1 + x**2)


def plot_integrand(bound):
    """Visualization of the integrand."""
    plt.close("integrand")
    fig, ax = plt.subplots(figsize=(7, 4), num="integrand")
    xs = np.linspace(-bound, bound, 5000)
```

(continues on next page)

(continued from previous page)

```
    ax.plot(xs, integrand_1d(xs))
    ax.set_xlim(-bound, bound)
    ax.set_title("Visualization of the integrand")
    ax.grid()
    ax.set_xlabel("$x$")
    ax.set_ylabel(r"$\cos(1/x)/(\pi(1 + x^2))$")


plot_integrand(1)
```




```
def demo_mc_1d(size):
    """Simple demonstration of MC."""
    rng = np.random.default_rng()
    xs = rng.standard_cauchy(size)
    return np.cos(1 / xs).mean()


demo_mc_1d(1000)
```

```
np.float64(0.36337628485330037)
```

```
def demo_quad_1d(size):
    """Integration with SciPy's quad function."""
    return quad(integrand_1d, -np.inf, np.inf, limit=size)


demo_quad_1d(1000)
```

```
(0.3678865226181152, 9.56198376034223e-06)
```

> The quad function complains about convergence, but all in all, it performs surprisingly well for such a difficult case.

12.2.1 Error estimation

Any Monte Carlo (MC) estimate is a stochastic quantity, simply because it is a function of (many) stochastic quantities, i.e. the sample points. Hence, the result is never exact.

Luckily, one can estimate the error quite easily. The variance of the Monte Carlo estimate is

$$\mathrm{VAR}\left[\frac{1}{N}\sum_{i=1}^{N} g(\mathbf{X}_i)\right] = \frac{1}{N^2}\sum_{i=1}^{N}\mathrm{VAR}\left[g(\mathbf{X}_i)\right] = \frac{1}{N}\mathrm{VAR}[g(\mathbf{X})]$$

In the first step, we utilized the propagation of variance for a linear combination of two stochastic variables:

$$\mathrm{VAR}[a\mathbf{X} + b\mathbf{Y}] = a^2\mathrm{VAR}[\mathbf{X}] + b^2\mathrm{VAR}[\mathbf{Y}] + 2ab\,\mathrm{COV}[\mathbf{X}, \mathbf{Y}]$$

where $\mathrm{COV}[\mathbf{X}, \mathbf{Y}]$ stands for the covariance of two stochastic quantities. No covariance is taken into account, because we assume independent sample points. The sum contains N times the same term because the samples $\mathbf{X}_i$ are drawn from the same distribution.

The standard error on the MC estimate is simply the square root of the variance:

$$\text{Std.Err.} = \sqrt{\frac{\mathrm{VAR}[g(\mathbf{X}_i)]}{N}}$$

The error decreases proportionally to $1/\sqrt{N}$ with increasing N. Hence, by taking a sufficiently large sample, the error can be made arbitrarily small.

Put differently, the number of required points is proportional to $1/(\text{Std.Err.})^2$. Note that this scaling is independent of the dimension of $\mathbf{X}$ and remains the same for high-dimensional integrals. This is very different from conventional quadrature methods, where the number of required points scales exponentially with the dimension of $\mathbf{X}$.

Numerical estimates of the standard error can be derived from an unbiased estimate of the variance:

$$\mathrm{VAR}\left[g(\mathbf{X}_i)\right] \approx \frac{1}{N-1}\sum_{i=1}^{N}\left(g(\mathbf{X}_i) - \overline{g(\mathbf{X}_i)}\right)^2$$

Just keep in mind that such estimates may not always be reliable and should not be trusted blindly.

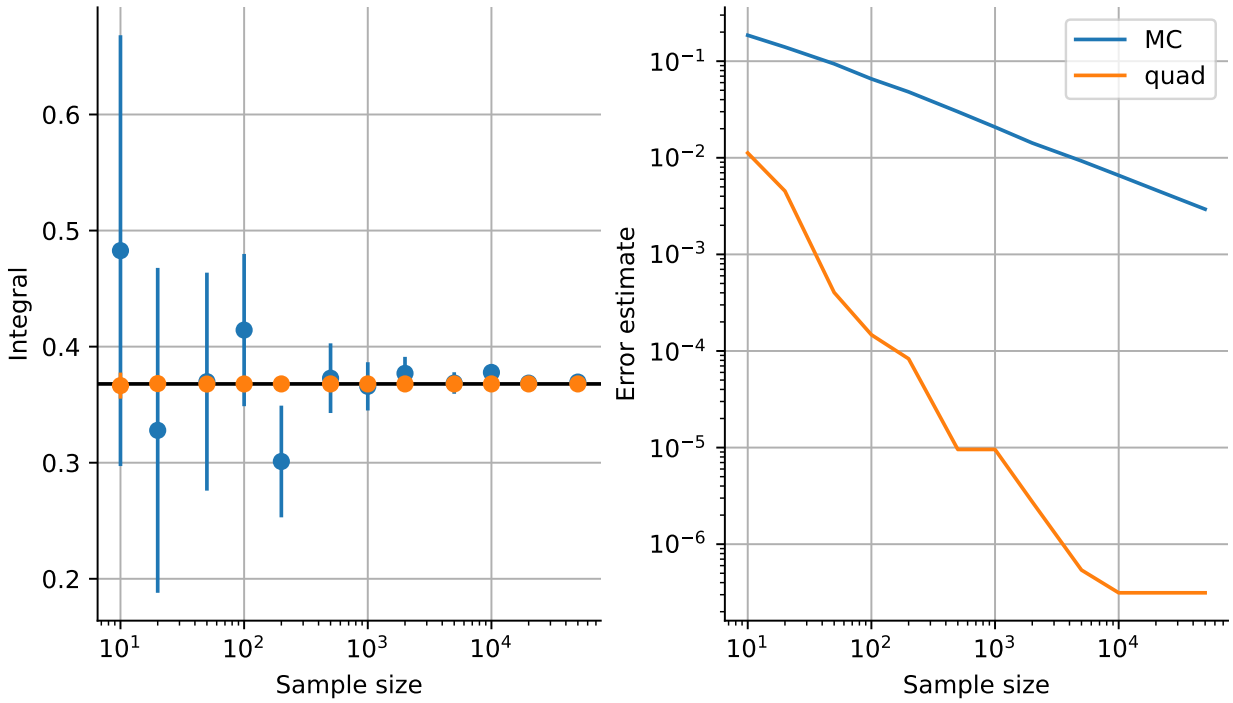
> **Error estimate for the one-dimensional integral**
>
> The code below computes the error estimate and shows the convergence with an increasing sample size.

```
def demo_mc_1d_error(size):
    rng = np.random.default_rng()
    x = rng.standard_cauchy(size)
    values = np.cos(1 / x)
    return values.mean(), values.std() / np.sqrt(size)


def plot_convergence_1d():
    plt.close("1d")
    sizes = np.array(
```

(continues on next page)

(continued from previous page)

```
        [10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000, 50000]
    )
    results_mc = np.array([demo_mc_1d_error(size) for size in sizes])
    results_quad = np.array([demo_quad_1d(size) for size in sizes])
    fig, axs = plt.subplots(1, 2, num="1d", figsize=(7, 4))

    def plot_integral(ax):
        ax.axhline(np.exp(-1), color="k")
        ax.errorbar(sizes, results_mc[:, 0], results_mc[:, 1], fmt="o")
        ax.errorbar(sizes, results_quad[:, 0], results_quad[:, 1], fmt="o")
        ax.set_xscale("log")
        ax.set_xlabel("Sample size")
        ax.set_ylabel("Integral")
        ax.grid()

    def plot_error(ax):
        ax.plot(sizes, results_mc[:, 1], label="MC")
        ax.plot(sizes, results_quad[:, 1], label="quad")
        ax.set_xscale("log")
        ax.set_yscale("log")
        ax.set_xlabel("Sample size")
        ax.set_ylabel("Error estimate")
        ax.grid()
        ax.legend(loc=0)

    plot_integral(axs[0])
    plot_error(axs[1])


plot_convergence_1d()
```



A few remarks about the results from the last example:

- The scaling of the MC error is clearly recognizable: for two additional orders in the number of points, the error decreases by one order.
- The scaling of the traditional quadrature error is much better, because this is a one-dimensional integral. When the number of dimensions increases, this quickly changes in favor of MC.

12.2.2 Application to error propagation

A straightforward application of MC is error propagation.

Consider any calculation for which the inputs, **X**, are uncertain, and you would like to estimate the uncertainties on the outcome.

One numerical solution to this problem is to repeat the calculation many times, each time drawing different random vaules from distributions that represent the uncertain inputs. This will result in a sample distribution of the calculation outcomes. All moments of the outcome samples can be analyzed, e.g. the mean and the standard deviation, the latter being a model of the uncertainty on the outcome. Note that all these moments are MC estimates with a mean and an uncertainty due to the finite number of sample points.

Example: error propagation in Torricelli's law

Torricelli observed the following in 1643:

> The velocity with which a liquid leaves the opening of a vessel is equal to
>
> $v = \sqrt{2gh}$
>
> where h is the height of the liquid level above the opening, and g is Standard gravity.

Let's assume that we are on the (fictitious) planet Zork and observe the following results after repeated measurements:

$$v \approx 3.20 \pm 0.86\,\mathrm{m/s}$$
$$h \approx 1.00 \pm 0.05\,\mathrm{m}$$

What is then the gravitational acceleration on Zork and the uncertainty in this result?

The following code cell shows how to compute the estimates and sampling uncertainties with MC.

```
def demo_torricelli(size):
    if size % 100 != 0:
        raise ValueError("Size must be a multiple of 100.")
    vs = np.random.normal(3.20, 0.86, size)
    hs = np.random.normal(1.00, 0.05, size)
    gs = vs**2 / (2 * hs)

    # Calculation of the mean and its uncertainty.
    eg = gs.mean()
    ug = gs.std() / np.sqrt(gs.size)
    print(f"Mean g on Zork [m/s^2]:  {eg:.3f} ± {ug:.3f}")

    # Calculation of the standard error and its uncertainty,
    # using batch size 100.
```

(continues on next page)

(continued from previous page)

```
    eeg = gs.reshape(-1, 100).std(axis=0).mean()
    ueg = gs.reshape(-1, 100).std(axis=0).std() / np.sqrt(size / 100)
    print(f"Error g on Zork [m/s^2]: {eeg:.3f} ± {ueg:.3f}")

    # Other properties can be estimate from the sample.
    ep = (gs > 9.81).mean() * 100
    up = (gs > 9.81).std() / np.sqrt(gs.size) * 100
    print("Probability that Zork has a higher gravitational constant than␣
↪Earth:")
    print(f"  {ep:.1f}% ± {up:.1f}%")


demo_torricelli(10000)
```

```
Mean g on Zork [m/s^2]:  5.501 ± 0.028
Error g on Zork [m/s^2]: 2.750 ± 0.023
Probability that Zork has a higher gravitational constant than Earth:
  7.6% ± 0.3%
```

Note that the values mentioned after the **± are uncertainties inherent to Monte Carlo** sampling. Larger samples will reduce these uncertainties for the same measurement. They are not related to the (propagation of) measurement errors.

The above example is only a simple demonstration. The same technique is applicable to a wide variety of more complicated calculations, e.g. weather forecasts, epidemiological models, rocket trajectories, investment portfolios, ray tracing, etc.

12.2.3 Common pitfall

For clarity, let's repeat the basic recipe of the Monte Carlo method:

$$\int g(\mathbf{x}) p_{\mathbf{X}}(\mathbf{x})\,\mathrm{d}\mathbf{x} \approx \frac{1}{N}\sum_{i=1}^{N} g(\mathbf{X}_i) = \overline{g(\mathbf{X})}$$

For some choices of g, it becomes infeasible to converge the MC estimate. In qualitative terms, this happens when:

- g is virtually zero in regions where $p_{\mathbf{X}}$ is significant, and
- g becomes very large when the probability density nearly vanishes.

For such a function g, outliers of the distribution will have a large contribution to the average. At best, this results in a large sampling error. At worst, there are no such outliers in the sample and the error goes unnoticed.

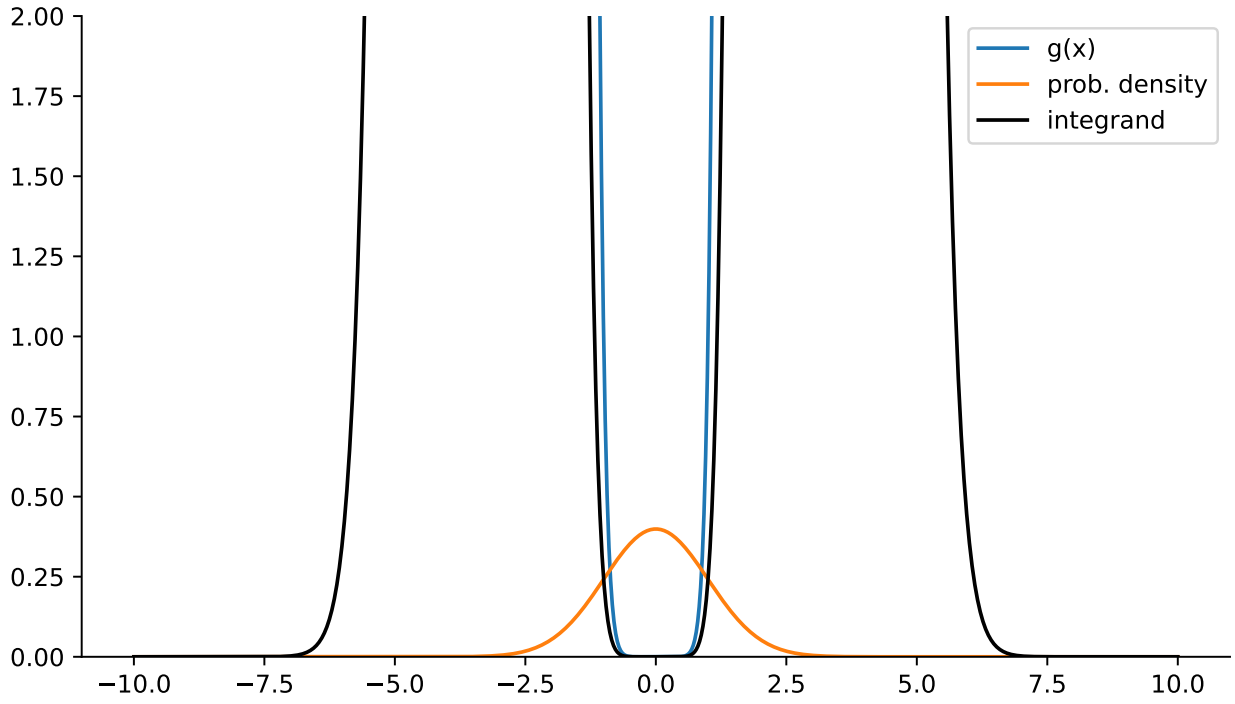
> **Example: an evil one-dimensional integral**
>
> Let's try to solve the following integral with MC:
>
> $$\int_{-\infty}^{+\infty} x^{10} \frac{\exp(-x^2/2)}{\sqrt{2\pi}}\,\mathrm{d}x = 945$$
>
> The integrand is the product of x^{10} and a standard normal probability density.
>
> The following code plots the integrand and the two separate factors.

```
def plot_integrand_evil():
    """Visualization of the integrand."""
```

(continues on next page)

(continued from previous page)

```
    plt.close("integrand_evil")
    fig, ax = plt.subplots(figsize=(7, 4), num="integrand_evil")
    xs = np.linspace(-10, 10, 5000)
    ax.plot(xs, xs**10, label="g(x)")
    ax.plot(xs, np.exp(-(xs**2) / 2) / np.sqrt(2 * np.pi), label="prob.␣
↪density")
    ax.plot(
        xs,
        xs**10 * np.exp(-(xs**2) / 2) / np.sqrt(2 * np.pi),
        "k",
        label="integrand",
    )
    ax.legend(loc=0)
    ax.set_ylim(0, 2)


plot_integrand_evil()
```



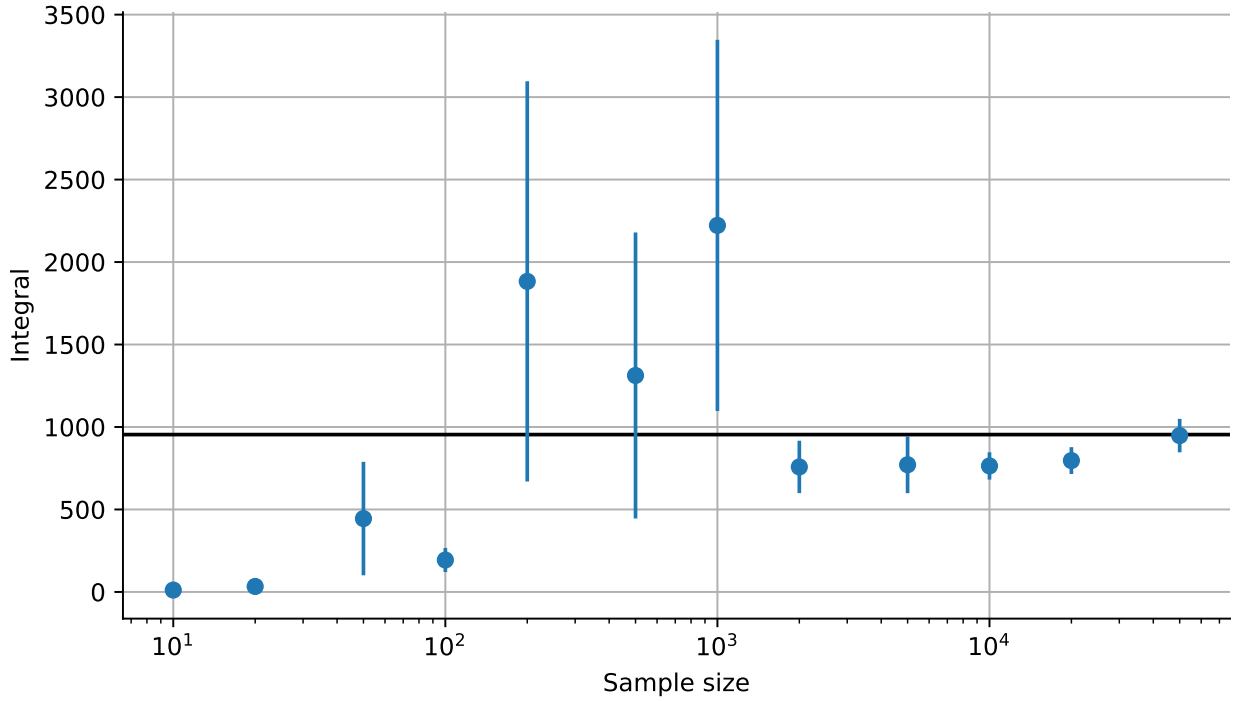
> This plot shows that the integrand is significant in regions where the probability density of drawing a sample point is very low. There is only a tiny probability of observing a large value of x^{10}. As a consequence, significant contributions to the integrand can easily be missed by drawing (only a few) random numbers from the standard normal distribution.
>
> The following cell visualizes the consequences: when using a small sample, the MC estimate of the integral can be very wrong (too low in this case). For the same reason, the uncertainty estimate will also be misleading in such cases.

```
def demo_mc_1d_evil_error(size):
    rng = np.random.default_rng()
```

(continues on next page)

(continued from previous page)

```
    x = rng.standard_normal(size)
    values = x**10
    return values.mean(), values.std() / np.sqrt(size)


def plot_convergence_1d_evil():
    plt.close("evil")
    fig, ax = plt.subplots(figsize=(7, 4), num="evil")
    sizes = np.array(
        [10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000, 50000]
    )
    results = np.array([demo_mc_1d_evil_error(size) for size in sizes])
    ax.axhline(954, color="k")
    ax.errorbar(sizes, results[:, 0], results[:, 1], fmt="o")
    ax.set_xscale("log")
    ax.set_xlabel("Sample size")
    ax.set_ylabel("Integral")
    ax.grid()


plot_convergence_1d_evil()
```



For the one-dimensional case, one can understand the problem intuitively, but for higher-dimensional integrals, intuition easily falls short and insufficient sampling is harder to recognize.

A more formal way of describing the problem is that MC has convergence issues when there is a large variance on $g(\mathbf{X}_i)$. The errors still scale proportionally with $1/\sqrt{N}$, but the prefactor, $\sqrt{\mathrm{VAR}[g(\mathbf{X}_i)]}$, is huge. In such cases, the uncertainty of the sampling estimate of this prefactor is also large (for the same reason), making it difficult to detect this pitfall empirically.

To solve this problem, one should construct another $g(\mathbf{X}_i)$, which reduces the risk of huge outliers with a small probability. This is not always straightforward.

Attempt to fix the evil example:

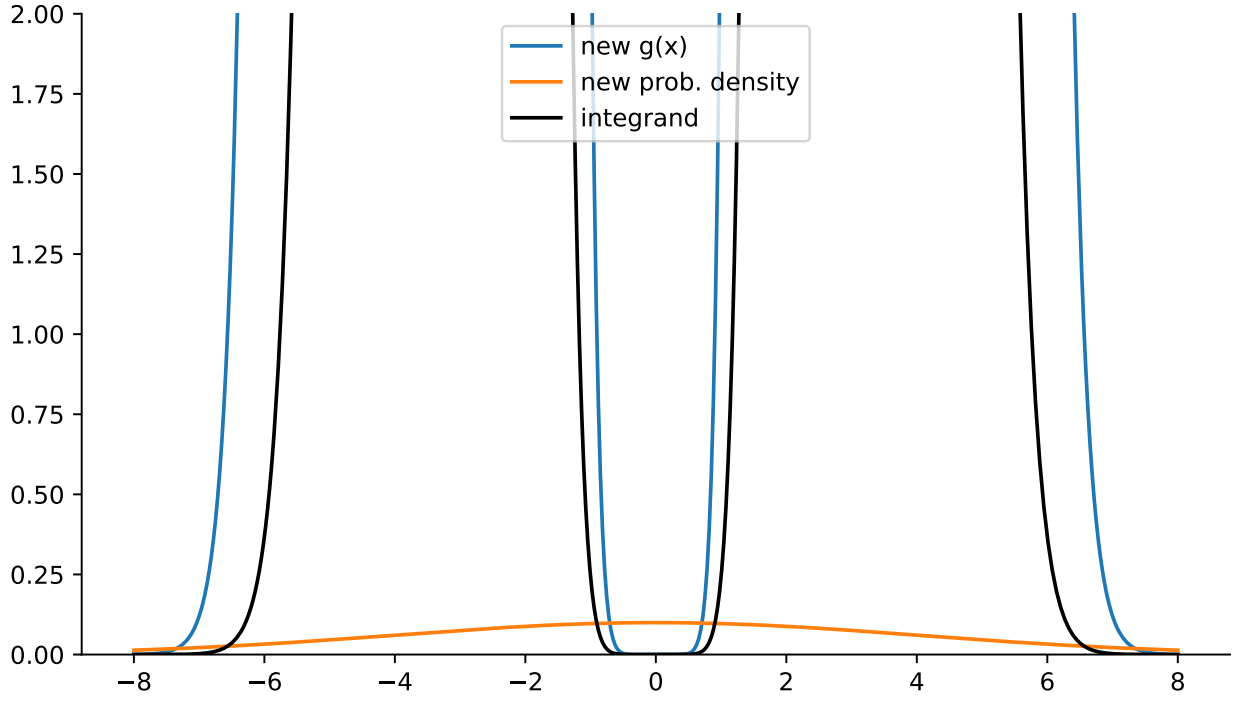
One may rewrite the integral as follows:

$$\int_{-\infty}^{+\infty} x^{10} a \exp\left(-\frac{a^2-1}{a^2}\frac{x^2}{2}\right) \frac{\exp(-(x/a)^2/2)}{a\sqrt{2\pi}} \mathrm{d}x = 945$$

With $a > 1$, the last factor becomes a broader normal distribution and the remainder of the integrand is suppressed for larger values of x, effectively limiting the pernicious behavior of x^{10}.

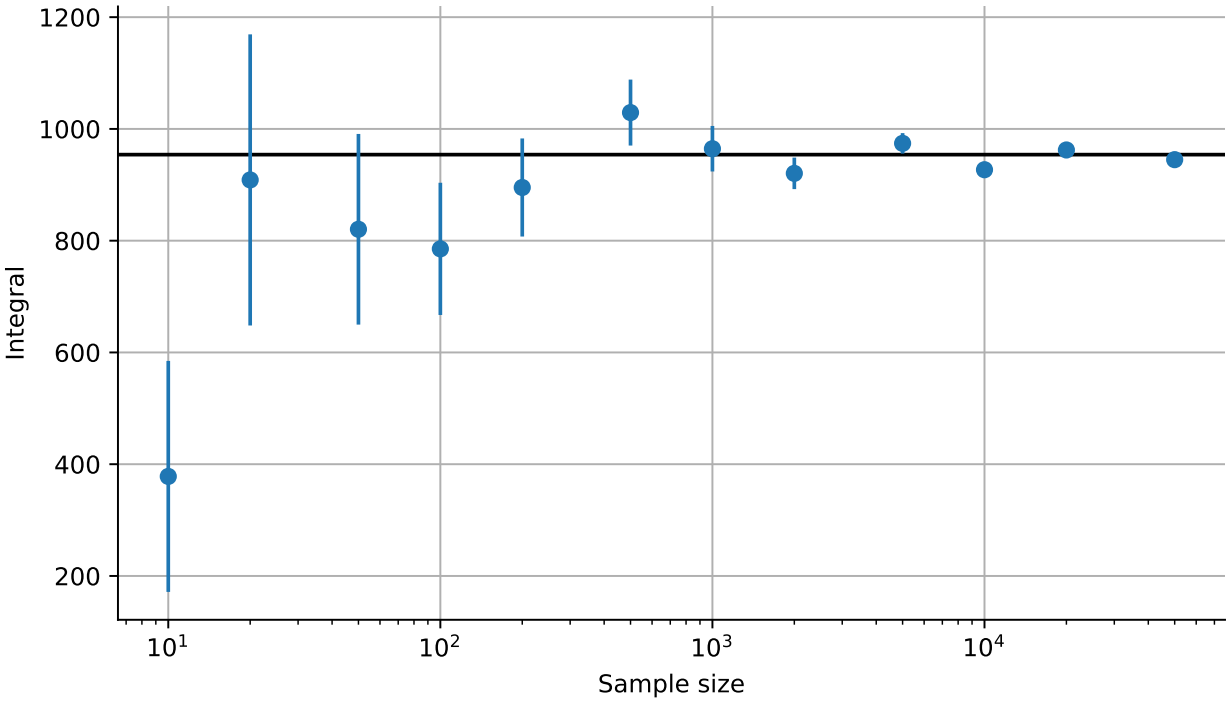
The following figure shows both factors in the integrand. At $a = 4$ the function g is only significant where the probability density is not negligible.

```
def plot_integrand_fixed(a=4):
    plt.close("integrand_fixed")
    fig, ax = plt.subplots(figsize=(7, 4), num="integrand_fixed")
    xs = np.linspace(-2 * a, 2 * a, 500)
    ax.plot(
        xs,
        xs**10 * a * np.exp(-(a**2 - 1) / a**2 * (xs**2 / 2)),
        label="new g(x)",
    )
    ax.plot(
        xs,
        np.exp(-((xs / a) ** 2) / 2) / np.sqrt(2 * np.pi) / a,
        label="new prob. density",
    )
    ax.plot(
        xs,
        xs**10 * np.exp(-(xs**2) / 2) / np.sqrt(2 * np.pi),
        "k",
        label="integrand",
    )
    ax.legend(loc=0)
    ax.set_ylim(0, 2)


plot_integrand_fixed()
```



> We can now use the rewritten integral to implement the MC method.

```
def demo_mc_1d_fixed_error(size, a=4):
    rng = np.random.default_rng()
    xs = rng.normal(0, a, size)
    values = xs**10 * a * np.exp(-(a**2 - 1) / a**2 * (xs**2 / 2))
    return values.mean(), values.std() / np.sqrt(size)


def plot_convergence_1d_fixed():
    plt.close("fixed")
    fig, ax = plt.subplots(figsize=(7, 4), num="fixed")
    sizes = np.array(
        [10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000, 50000]
    )
    results = np.array([demo_mc_1d_fixed_error(size) for size in sizes])
    ax.axhline(954, color="k")
    ax.errorbar(sizes, results[:, 0], results[:, 1], fmt="o")
    ax.set_xscale("log")
    ax.set_xlabel("Sample size")
    ax.set_ylabel("Integral")
    ax.grid()


plot_convergence_1d_fixed()
```



> The numerical integration improves, and also the error estimates become reliable again.

Broadening or modifying the distribution in the integral is a common technique with different names, depending on the application. Methods like *importance sampling*, *reweighting*, *biased sampling*, … are all based on the same concept. You are free to choose which probability density to use, and you can always compensate for that choice in the function g.

> **Note.** More traditional quadrature methods would be more appropriate for the evil example. However, they would be of no use when similar difficulties arise in high-dimensional integrals.

12.2.4 Monte Carlo integration

The name **Monte Carlo integration** is often used for the following special case:

$$I = \int_\Omega g(\mathbf{x})\,\mathrm{d}\mathbf{x}$$

where, at first sight, the probability density is missing. The integral runs over a finite domain Ω instead of over the whole space.

One may always insert a uniform distribution and divide out its normalization:

$$I = V_\Omega \int_\Omega g(\mathbf{x}) p_{\mathbf{X},\mathrm{uniform}}(\mathbf{x})\,\mathrm{d}\mathbf{x}$$

where V_Ω, the volume of the domain, is also the inverse of the normalization constant of the uniform distribution, $p_{\mathbf{X},\mathrm{uniform}}(\mathbf{x})$. In this form, the Monte Carlo method described above is applicable.

Monte Carlo integration thus is essentially a quadrature method with equal weights and randomized grid points. It is numerically well-behaved as long as g varies smoothly. In line with numerical quadrature in general, MC integration may become problematic when g is negligible nearly everywhere in the domain. For such ill-posed cases, adaptive methods such as the MISER algorithm have been developed to focus on subdomains where g is more informative.

Example

An easy example of Monte Carlo integration estimates π.

The domain Ω in this example is a unit square: $x_0 \in [0, 1]$ and $x_1 \in [0, 1]$, for which $V_\Omega = 1$. The function $g(\mathbf{x})$ is 1 for all points within a distance of 1 from the origin and 0 outside:

$$g(\mathbf{x}) = H(1 - x_0^2 - x_1^2),$$

where H is the Heaviside step function. The integral of g over Ω is then the quarter of the area of a unit circle, i.e. $\pi/4$:

$$\begin{aligned}\frac{\pi}{4} &= \int_0^1 \mathrm{d}x_0 \int_0^1 \mathrm{d}x_1 \, g(x_0, x_1) \\ &= V_\Omega \int_0^1 \mathrm{d}x_0 \int_0^1 \mathrm{d}x_1 \, g(x_0, x_1) \, p_{X_0 X_1, \text{uniform}}(x_0, x_1) \\ &\approx \frac{1}{N} \sum_{k=1}^{N} g(\mathbf{X}_k)\end{aligned}$$

This shows that $\pi/4$ can be approximated as the ratio of the number of samples that fall inside the circle over the total number of samples, N.

The code and plot for this example is given below.

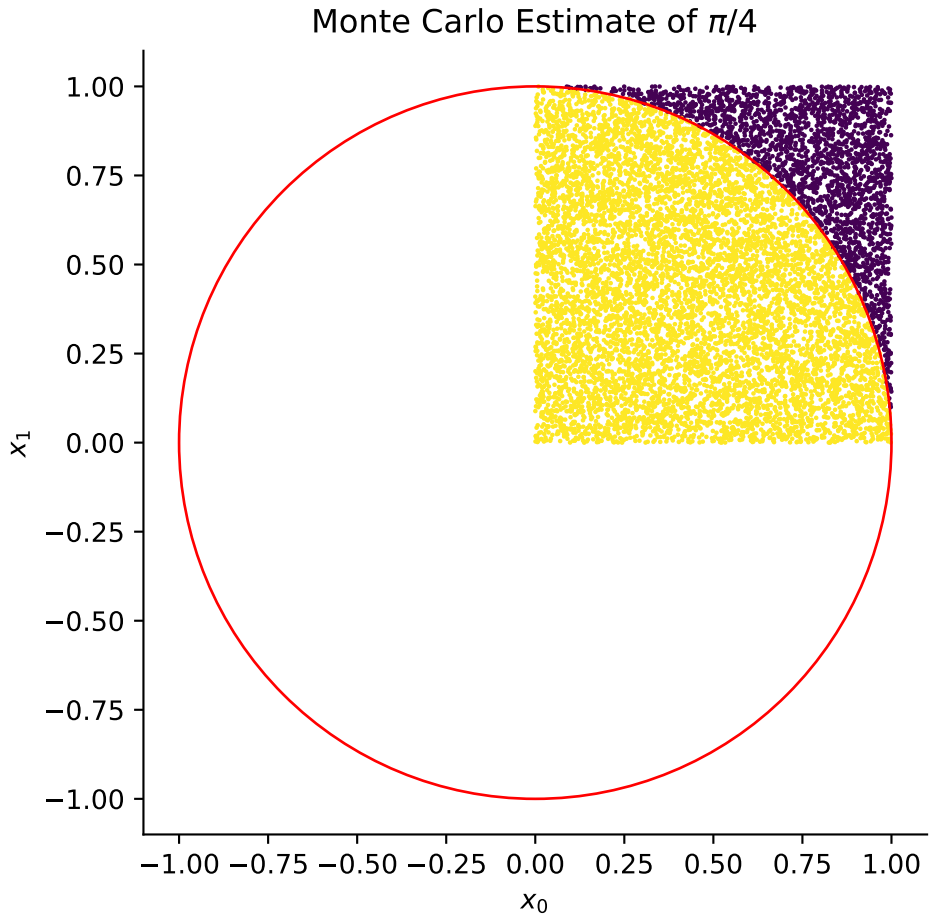
```
def demo_pi_mc(n):
    """Estimate the value of π using Monte Carlo integration.

    Parameters
    ----------
    n : int
        Number of random points to generate.
    """

    # Generate random points (x, y) inside the unit square
    random_points = np.random.rand(n, 2)

    # Check which points are inside the quarter circle (x^2 + y^2 <= 1)
    inside_circle = np.sum(random_points**2, axis=1) <= 1

    # Estimate π using the ratio of samples inside the circle over the total.
    estimate_pi_over_four = np.sum(inside_circle) / n

    # Visualize the points and the quarter circle
    plt.close("pi_mc")
    fig, ax = plt.subplots(num="pi_mc")
    ax.scatter(
        random_points[:, 0],
        random_points[:, 1],
        c=inside_circle,
        cmap="viridis",
        s=0.5,
    )
    circle = plt.Circle((0, 0), 1, color="red", fill=False)
    ax.add_patch(circle)
```

(continues on next page)

(continued from previous page)

```
    ax.set_title(r"Monte Carlo Estimate of $\pi/4$")
    ax.set_xlabel("$x_0$")
    ax.set_ylabel("$x_1$")
    ax.set_aspect("equal")

    print(f"Monte Carlo Estimate of π/4: {estimate_pi_over_four:8.5f}")
    print(f"Monte Carlo Estimate of π:   {4 * estimate_pi_over_four:8.5f}")


demo_pi_mc(10000)
```

```
Monte Carlo Estimate of π/4:  0.77280
Monte Carlo Estimate of π:    3.09120
```



12.3 Discrete Markov chain Monte Carlo methods

In the previous section, the stochastic variables could be sampled with random number generators in NumPy. Sometimes, functions of these random numbers had to be computed. For practically all univariate distributions, uniform samples can be transformed, e.g. with inverse transform sampling. Such transformations are not always practical, and are usually impossible for complicated multivariate distributions.

In this section, we'll discuss Markov chain Monte Carlo (MCMC), which can be used to draw samples from any distribution, even when the methods above fall short. To use the MCMC method, only the probability density up to a constant factor must be known.

This section covers the very basics of discrete Markov chain Monte Carlo. The field has been under development since the early days of electronic computers and is a vast topic by itself. The book of David MacKay is a classic reference (written from a statistics perspective).

12.3.1 Definition of a discrete Markov chain

A **discrete Markov chain** is a stochastic process, i.e. a sequence of stochastic quantities $\{\mathbf{X}_k\}$, in which the probability of observing $\mathbf{x}_{i+1}$ is only determined by:

1. the previous state x_i, and
2. the index i.

Formally, one may write

$$p_{\mathbf{X}_{i+1}}(\mathbf{x}_{i+1}) = \int T_{i\to i+1}(\mathbf{x}_{i+1}|\mathbf{x}_i)\, p_{\mathbf{X}_i}(\mathbf{x}_i)\, \mathrm{d}\mathbf{x}_i$$

One can interpret $T_{i\to i+1}$ as a conditional probability density, and it is often called the "transition probability density".

Some remarks:

- It is important that the function $T_{i\to i+1}$ is *not explicitly dependent on older states*, such as $\mathbf{x}_{i-1}$. Such a dependency would result in a non-Markov chain.
- When the function $T_{i\to i+1}$ is the same for all i, the chain is called *time-homogeneous* and one may just write T. For ease of notation, this convention will always be followed below.
- The adjective *discrete* means that the states are labeled by a discrete index i. In continuous Markov chains, the states are labeled by a continuous variable, e.g. a time t.

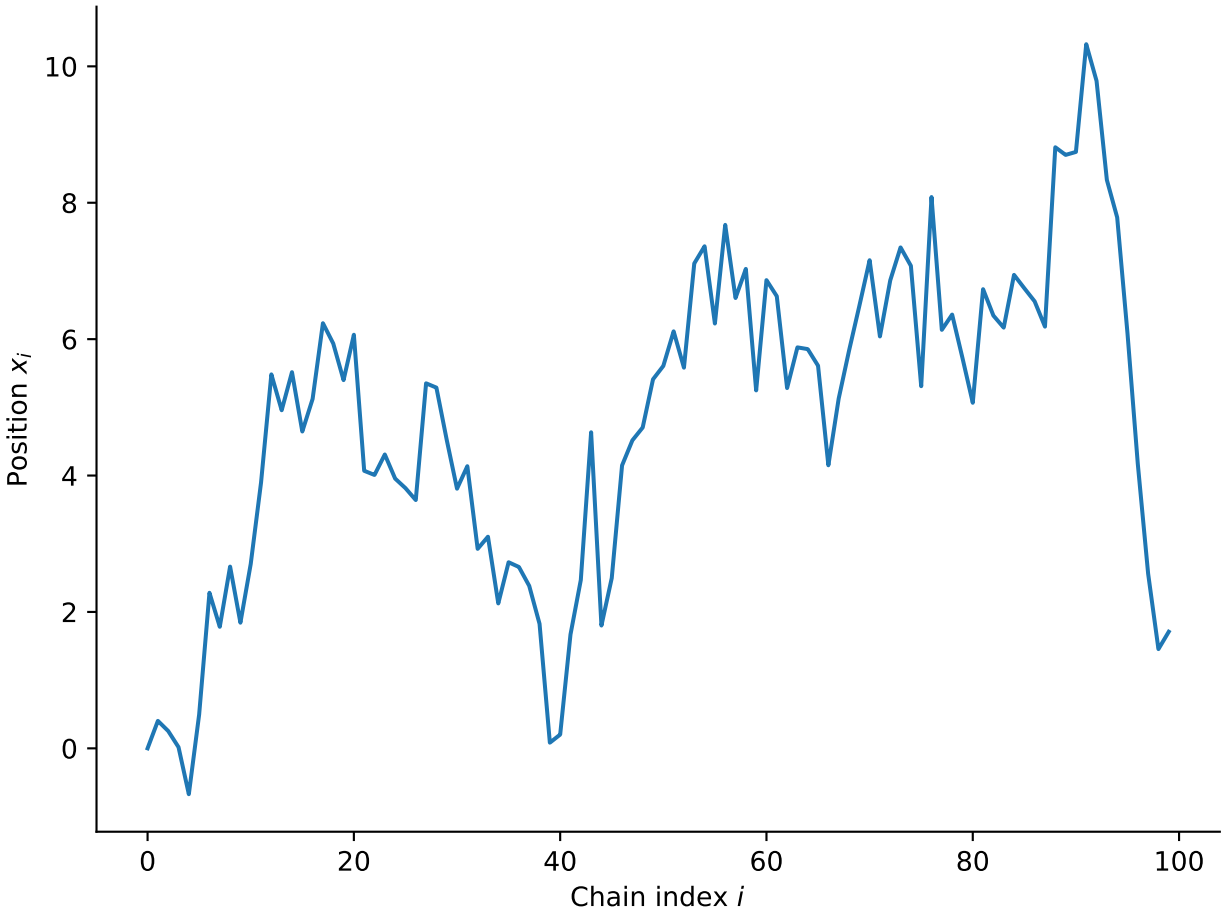
An **important property** of the transition probability is the following normalization:

$$\int T(\mathbf{x}_{i+1}|\mathbf{x}_i)\, \mathrm{d}\mathbf{x}_{i+1} = 1$$

> **Proof.** One should simply require that the probability density of state $i+1$ is properly normalized when state i is a Dirac delta distribution.
>
> $$1 = \int p_{\mathbf{X}_{i+1}}(\mathbf{x}_{i+1})\, \mathrm{d}\mathbf{x}_{i+1} = \iint T(\mathbf{x}_{i+1}|\mathbf{x}_i)\, \delta(\mathbf{x}_i^0 - \mathbf{x}_i)\, \mathrm{d}\mathbf{x}_{i+1}\, \mathrm{d}\mathbf{x}_i = \int T(\mathbf{x}_{i+1}|\mathbf{x}_i^0)\, \mathrm{d}\mathbf{x}_{i+1}$$
>
> Note that this also shows how to sample a Markov chain in practice. At the point that we have a sample point $\mathbf{x}_i^0$, its position is fixed, as described by the Dirac delta function, and the next point is simple generated by sampling the transition probability in which $\mathbf{x}_i^0$ appears as a parameter.

> **Simple example: a random walk.**
>
> Consider a one-dimensional particle with position X_i in state i. Between two states, the particle can make a stochastic step, sampled from a unit normal distribution:
>
> $$T(x_{i+1}|x_i) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{(x_{i+1}-x_i)^2}{2}\right)$$
>
> The function below visualizes such a random walk, which represents one sample point of the entire Markov chain. (Note that subsequent states in the chain are statistically correlated.)

```
def plot_random_walk(size=100):
    # Generate the chain.
    rng = np.random.default_rng()
    states = np.zeros(size)
    for i in range(1, size):
        states[i] = states[i - 1] + rng.standard_normal()

    # Plot the chain
    plt.close("random_walk")
    fig, ax = plt.subplots(num="random_walk")
    ax.plot(states)
    ax.set_xlabel("Chain index $i$")
    ax.set_ylabel("Position $x_i$")


plot_random_walk()
```



> **Remark**: all PRNGs behave like Markov chains, except that they are only pseudo-random. They share the property that the next state is only determined by the current state.

12.3.2 The stationary distribution of a discrete Markov chain

As mentioned above, we assume that the Markov chain is *time-homogeneous*. All transition are described by the same transition probability T.

A stationary distribution has the following property:

$$\int T(\mathbf{x}_2|\mathbf{x}_1)p(\mathbf{x}_1)\,\mathrm{d}\mathbf{x}_1 = p(\mathbf{x}_2)$$

In words, the transition to the next state leads to the same probability density.

One can therefore consider **the chain as a random number generator for its corresponding stationary distribution**. It can be used as follows:

1. Draw an initial sample point with a non-zero probability in the stationary distribution: $\mathbf{x}_1^0$. In this case, one can claim that the initial sample point is drawn from the stationary distribution, even if it is improbable.
2. Generate a next sample point, using the transition probability $T(\mathbf{x}_2|\mathbf{x}_1^0)$. Because the previous sample point could have been drawn from the stationary distribution, the current sample is also a valid sample point.
3. Repeat step 2, now with $T(\mathbf{x}_3|\mathbf{x}_2^0)$, $T(\mathbf{x}_4|\mathbf{x}_3^0)$, … until you have enough samples.

We will not elaborate on the statistical technicalities here, but the following are worth mentioning:

- The stationary distribution **does not always exist**.

  > For example, in the random walk example, the distribution will always become broader and flatter, without ever reaching a stationary regime. This behavior is recovered for any initial distribution.

- When a stationary distribution exists, it is **not necessarily unique**.

  > Consider for example a random walk that is only allowed within two intervals $[-3, -1]$ and $[1, 3]$. Uniform distributions in either interval are stationary, as well as a uniform distribution over both intervals at the same time.

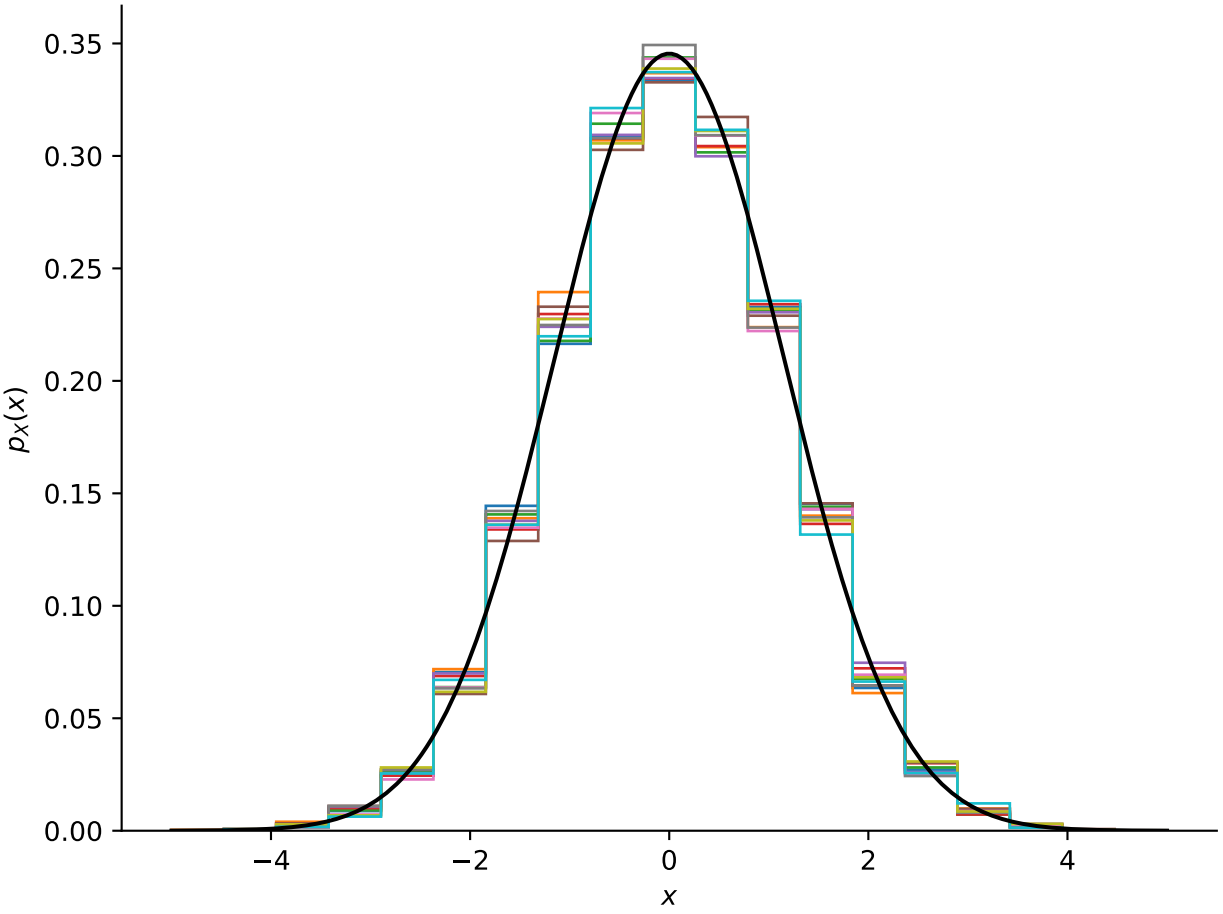
> **Example 1**
>
> Consider the following transition probability
>
> $$T(x_{i+1}|x_i) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{(x_{i+1} - x_i/2)^2}{2}\right)$$
>
> This means the previous position is scaled by a factor 1/2 and then randomly displaced by a standard-normally distributed step.
>
> One may derive the stationary distribution analytically for this case. It is also a normal distribution with mean 0 and standard deviation $\sqrt{4/3}$. (The derivation is a nice statistics exercise.)
>
> The following code illustrates that the stationary distribution is not affected by the initial state:

```
def plot_stationary_example(size=10000):
    rng = np.random.default_rng()
    bins = np.linspace(-5, 5, 20)
    plt.close("stationary_example")
    fig, ax = plt.subplots(num="stationary_example")
    for _ in range(10):
```

(continues on next page)

(continued from previous page)

```
        # Generate the chain
        xs = np.zeros(size)
        xs[0] = rng.normal(-10, 10)
        for i in range(1, size):
            # xs[i] = xs[i - 1] / 2 + rng.uniform(-1, 1)
            xs[i] = xs[i - 1] / 2 + rng.normal(0, 1)

        # Plot the histogram
        ax.hist(xs, bins, histtype="step", density=True)

    # Plot the stationary distribution.
    sigma = np.sqrt(4 / 3)
    xs = np.linspace(-5, 5, 200)
    ax.plot(
        xs,
        np.exp(-((xs / sigma) ** 2) / 2) / (sigma * np.sqrt(2 * np.pi)),
        color="k",
    )
    ax.set_xlabel("$x$")
    ax.set_ylabel("$p_X(x)$")


plot_stationary_example()
```



Example 2: Eigenvalue problem

Finding the stationary distribution can also be seen as an eigenvalue problem. To clarify this, consider an example of a discrete system that can be in of three states $s \in \{0, 1, 2\}$, as opposed to a continues state x in the rest of this section. The probability to move from state s_i to s_{i+1} is given by the transition matrix $T_{i+1,i}$:

$$T = \begin{bmatrix} 0.6 & 0.3 & 0.1 \\ 0.3 & 0.2 & 0.5 \\ 0.1 & 0.5 & 0.4 \end{bmatrix}$$

If the stationary distribution exists and is unique, many applications T to an initial state p° will converge toward p:

$$p = \lim_{N \to \infty} T^N p^\circ$$

This is infact a form of power iteration with eigenvalue 1.

In line with the theory above, the stationary distribution is described by a vector $p \in \mathbb{R}^3$ that satisfies the following equation:

$$Tp = p$$

This means that the stationary distribution is an eigenvector with eigenvalue 1, which is consistent with the power iteration. We compute the eigenvector p of $\lambda = 1$ in the following code:

```
def demo_markov_chain():
    # Define the transition matrix
    T = np.array([[0.6, 0.3, 0.1], [0.3, 0.2, 0.5], [0.1, 0.5, 0.4]])

    # Set the initial distribution
    probabilities = np.array([1, 0, 0])

    # Simulate 50 iteration steps to find the stationary distribution
    for _ in range(50):
        probabilities = T @ probabilities
    print("The stationary probabilities are:\n  ", probabilities)

    # Compute eigenvalues and the eigenvector corresponding to the eigenvalue␣
↪1.
    eigenvalues, eigenvectors = np.linalg.eigh(T)
    print("The eigenvalues are:\n  ", eigenvalues)

    # Check if there is a unique eigenvalue = 1, within some threshold.
    eps = 1e-5
    is_one = abs(eigenvalues - 1) < eps
    if any(is_one):
        # Find the corresponding stationary eigenvector
        idx_one = is_one.nonzero()[0][0]
        stationary_vector = eigenvectors[:, idx_one]
        print("The eigenvector with eigenvalue 1 is:\n  ", stationary_vector)
        stationary_vector /= stationary_vector.sum()
        print("After L1 normalization:\n  ", stationary_vector)
    else:
        print("No unique eigenvalue of 1 found.")
```

(continues on next page)

(continued from previous page)

```
demo_markov_chain()
```

```
The stationary probabilities are:
   [0.33333333 0.33333333 0.33333333]
The eigenvalues are:
   [-0.24641016  0.44641016  1.        ]
The eigenvector with eigenvalue 1 is:
   [-0.57735027 -0.57735027 -0.57735027]
After L1 normalization:
   [0.33333333 0.33333333 0.33333333]
```

> As we can see after some time there is an equal chance of being in any of the 3 states!
>
> Markov chains can be used as a random number generator for its corresponding stationary distribution. In this case the stationary distribution is quite simple (the uniform distribution), but for more complex distributions this is very helpful. Below we show a random number generator that draws from the distribution corresponding to the stationary state.

```
def demo_random_sampling():
    # Define the transition matrix
    T = np.array([[0.6, 0.3, 0.1], [0.3, 0.2, 0.5], [0.1, 0.5, 0.4]])

    # Start the chain with an initial state.
    chain = [0]

    # Extend the chain, each time use the transition probability
    # to sample the next point.
    for _ in range(100):
        state_previous = chain[-1]
        state_next = np.random.choice([0, 1, 2], p=T[state_previous])
        chain.append(state_next)
    chain = np.array(chain)

    print("100 random numbers:\n", chain)
    print("Number of 0s:", (chain == 0).sum())
    print("Number of 1s:", (chain == 1).sum())
    print("Number of 2s:", (chain == 2).sum())


demo_random_sampling()
```

```
100 random numbers:
 [0 1 2 2 2 1 0 1 2 1 0 0 0 1 2 1 0 0 0 0 1 1 0 0 0 1 2 2 0 0 0 2 1 1 0 0 0
 0 0 1 1 0 2 2 2 2 1 1 1 2 1 1 1 0 1 2 1 0 0 0 0 0 0 1 2 2 2 1 2 1 2 1 2 0
 2 2 1 2 2 2 2 2 1 2 1 0 0 1 2 0 1 2 1 2 2 1 2 2 2 0 0]
Number of 0s: 34
Number of 1s: 32
Number of 2s: 35
```

Remarks:

1. Each state appears with approximately the same probability, as expected.
2. The states exhibit time-correlation. This is expected from the probability matrix: once in state zero, there is a 60% chance to go to zero again, etc.

12.3.3 Designing a Markov chain for any stationary distribution

Finding the stationary distribution of a chain is generally challenging. However, for a given stationary distribution, there are an uncountable infinite number of Markov chains. In practice, it is even fairly straightforward to construct a Markov chain for any given stationary distribution. Once the Markov chain is defined, sampling the stationary distribution is trivial, as explained in the previous section.

The most general approach for constructing a suitable Markov chain is the Metropolis-Hastings algorithm. Many other methods can be seen as special cases of this general framework.

12.3.4 Metropolis-Hastings algorithm

Global Balance versus Detailed Balance

- A stationary distribution satisfies the "global balance" condition, that is, after convolution with the transition probability, the same probability is recovered.

$$\int T(\mathbf{x}_2|\mathbf{x}_1)p(\mathbf{x}_1)\,\mathrm{d}\mathbf{x}_1 = p(\mathbf{x}_2)$$

  This condition does not impose much structure on T and leaves plenty of freedom to decide from where to where the transition probability displaces sample points.

- A more restrictive requirement is called "detailed balance":

$$T(\mathbf{x}_2|\mathbf{x}_1)p(\mathbf{x}_1) = T(\mathbf{x}_1|\mathbf{x}_2)p(\mathbf{x}_2)$$

  This can be interpreted as follows: for a stationary distribution, the following two probabilities must be equal:

  – The probability density of finding a sample point at $\mathbf{x}_1$ and moving it to $\mathbf{x}_2$.

  – The probability density of finding a sample point at $\mathbf{x}_2$ and moving it to $\mathbf{x}_1$.

One may also show that detailed balance is a sufficient condition for global balance.

Proof:

$$\int T(\mathbf{x}_2|\mathbf{x}_1)p(\mathbf{x}_1)\,\mathrm{d}\mathbf{x}_1 = \int T(\mathbf{x}_1|\mathbf{x}_2)p(\mathbf{x}_2)\,\mathrm{d}\mathbf{x}_1 = p(\mathbf{x}_2)\int T(\mathbf{x}_1|\mathbf{x}_2)\,\mathrm{d}\mathbf{x}_1 = p(\mathbf{x}_2)$$

Derivation

The Metropolis-Hastings algorithm defines a Markov chain that satisfies detailed balance for any given stationary distribution.

The transitions in Metropolis-Hastings Markov chain are constructed in two steps:

- **Generation step.** A displacement of $\mathbf{x}_1$ to $\mathbf{x}_2$ is generated with a *proposal* distribution $g(\mathbf{x}_2|\mathbf{x}_1)$.

> For example, this can be the same normally distributed step size as in the random-walk example above.

- **Acceptance or rejection step.** After constructing $\mathbf{x}_2$ from $\mathbf{x}_1$, it is further assessed and accepted with a probability $A(\mathbf{x}_2, \mathbf{x}_1)$. If not accepted, the next state is identical to the current.

> The exact form of the acceptance probability will be specified later. It is the crucial component of the algorithm.

The total transition probability is the product of these two probabilities: $T(\mathbf{x}_2|\mathbf{x}_1) = g(\mathbf{x}_2|\mathbf{x}_1)A(\mathbf{x}_2, \mathbf{x}_1)$.

To find *a* correct expression for the acceptance probability, detailed balance is imposed:

$$g(\mathbf{x}_2|\mathbf{x}_1)A(\mathbf{x}_2, \mathbf{x}_1)p(\mathbf{x}_1) = g(\mathbf{x}_1|\mathbf{x}_2)A(\mathbf{x}_1, \mathbf{x}_2)p(\mathbf{x}_2)$$

and solved towards the ratio of acceptance probabilities:

$$\frac{A(\mathbf{x}_2, \mathbf{x}_1)}{A(\mathbf{x}_1, \mathbf{x}_2)} = \frac{p(\mathbf{x}_2)}{p(\mathbf{x}_1)} \frac{g(\mathbf{x}_1|\mathbf{x}_2)}{g(\mathbf{x}_2|\mathbf{x}_1)}$$

The highest possible acceptance probabilities satisfying this equation are

$$A(\mathbf{x}_2, \mathbf{x}_1) = \min\left(1, \frac{p(\mathbf{x}_2)}{p(\mathbf{x}_1)} \frac{g(\mathbf{x}_1|\mathbf{x}_2)}{g(\mathbf{x}_2|\mathbf{x}_1)}\right)$$

This expression becomes more intuitive when the *proposal* distributions become symmetric, i.e. $g(\mathbf{x}_1|\mathbf{x}_2) = g(\mathbf{x}_2|\mathbf{x}_1)$. In that case, one gets:

$$A(\mathbf{x}_2, \mathbf{x}_1) = \min\left(1, \frac{p(\mathbf{x}_2)}{p(\mathbf{x}_1)}\right)$$

This means the following:

- A transition to higher probability density is always accepted.
- A transition to lower probability density is accepted with a probability $p(\mathbf{x}_2)/p(\mathbf{x}_1)$.

To apply this method, one must merely be able to compute the probability density up to an unknown normalization factor. In practically all applications, this is possible.

Algorithm

Building on the above derivation, the Metropolis-Hastings algorithm works as follows:

1. Start with an initial state for which the (stationary) probability is non-zero.
2. Generate a step according to the proposal distribution $g(\mathbf{x}_2|\mathbf{x}_1)$.
3. Compute the acceptance ratio

   $$\mathrm{AR} = \frac{p(\mathbf{x}_2)}{p(\mathbf{x}_1)} \frac{g(\mathbf{x}_1|\mathbf{x}_2)}{g(\mathbf{x}_2|\mathbf{x}_1)}$$

   If this is larger than one, accept the step. If it is smaller than one, accept the step with probability AR. If the step is not accepted, the new state becomes equal to the old state.
4. Repeat steps 2 and 3 until the sample size is sufficient.

12.3.5 Simple examples of Metropolis-Hastings Markov chains

The following function is a generic MH Markov chain implementation. It will be used by the examples below.

```
def mhmc_driver(size, xinit, ln_stat_dens, prop_gen, ln_prop_dens=None,␣
↪rng=None):
    """Sample a Metropolis--Hastings Markov chain.

    Parameters
    ----------
    size
        The number of samples to generate.
    xinit
        The initial state.
        Any type is allowed, as long as it is understood by the
        following three parameters, which are all functions.
    ln_stat_dens
        A function evaluating the logarithm of the stationary
        probability density of interest, up to an unknown
        normalization factor.
        It takes one argument (same type as xinit) and returns
        a floating point number.
    prop_gen
        A generator for the proposal distribution. It takes
        the current state as an argument, and it returns a proposed
        new state. Both argument and return value are of the
        same type as xinit.
    ln_prop_dens
        A function evaluating the logarithm of the proposal density.
        It takes two arguments: the final and initial state (in that
        order, both of the same type as xinit) and returns
        the probability density of the proposal.
        When not given, the proposal density is assumed to be
        symmetric.
    rng
        A random number generator, must have the uniform method.

    Returns
    -------
    chain
        A list of states.
    ln_sds
        Logarithm of the stationary probability density for each
        state in the chain.

    """
    if rng is None:
        rng = np.random.default_rng()
    # Initialize algorithm.
    xcur = xinit
    ln_sd_cur = ln_stat_dens(xcur)
    # Lists in which the output is stored.
    chain = [xcur]
    ln_sds = [ln_sd_cur]
    while len(chain) < size:
        # 1. Proposal
        xnew = prop_gen(xcur)

        # 2. Compute logarithm of acceptance ratio
```

(continues on next page)

(continued from previous page)

```
        ln_sd_new = ln_stat_dens(xnew)
        ln_ratio = ln_sd_new - ln_sd_cur
        if ln_prop_dens is not None:
            ln_ratio += ln_prop_dens(xcur, xnew) - ln_prop_dens(xnew, xcur)

        # 3. Accept or reject
        accept = ln_ratio > 0 or np.exp(ln_ratio) > rng.uniform(0, 1)
        if accept:
            xcur = xnew
            ln_sd_cur = ln_sd_new
        chain.append(xcur)
        ln_sds.append(ln_sd_cur)

    return chain, ln_sds
```

Numerical 1D demonstration

Consider a simple probability density with the shape of a cosine function.

$$p_X(x) \propto \begin{cases} \cos(x) & \text{if} \quad |x| \leq \pi/2 \\ 0 & \text{if} \quad |x| > \pi/2 \end{cases}$$

Note that we do not bother normalizing the density. (The norm of the given form is 2, which is used for the plot below.)

The integral we would like to compute is:

$$\int_{-\pi/2}^{\pi/2} x^2 \frac{\cos(x)}{2} \, \mathrm{d}\mathbf{x} = \frac{\pi^2}{4} - 2 \approx 0.467401100272340$$

The following cell demonstrates how the MH algorithm can sample this distribution, and how the integral is calculated. A uniform proposal distribution is used.

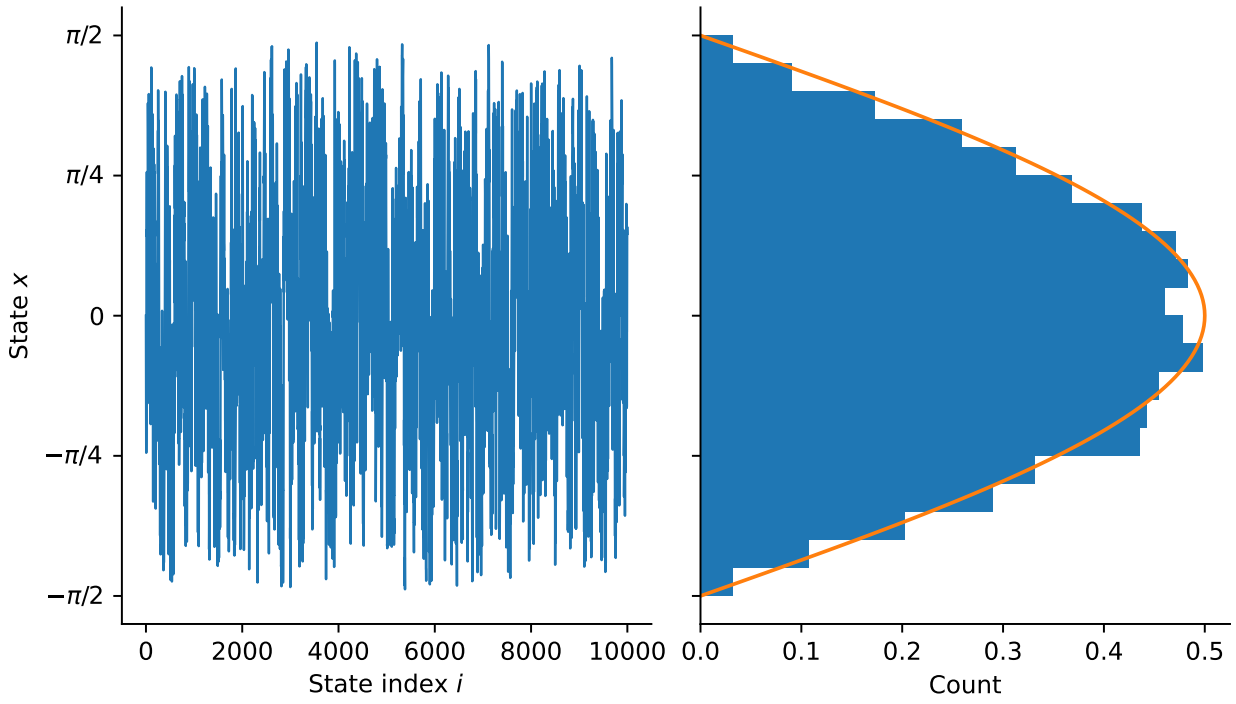
```
def demo_cosine():
    rng = np.random.default_rng()

    def prop_gen(xcur):
        return xcur + rng.uniform(-0.5, 0.5)

    def ln_stat_dens(x):
        if abs(x) > np.pi / 2:
            return -np.inf
        return np.log(np.cos(x))

    chain, _ = mhmc_driver(10000, 0.0, ln_stat_dens, prop_gen)

    # Plot the chain and the histogram including the analytical distribution
    plt.close("cosine")
    fig, axs = plt.subplots(1, 2, figsize=(7, 4), num="cosine", sharey=True)
    axs[0].plot(chain, lw=1)
    axs[0].set_xlabel("State index $i$")
    axs[0].set_ylabel("State $x$")
    axs[0].set_yticks(
        [-np.pi / 2, -np.pi / 4, 0, np.pi / 4, np.pi / 2],
        [r"$-\pi/2$", r"$-\pi/4$", "0", r"$\pi/4$", r"$\pi/2$"],
    )
```

(continues on next page)

(continued from previous page)

```
    axs[1].hist(
        chain,
        np.linspace(-np.pi / 2, np.pi / 2, 21),
        orientation="horizontal",
        density=True,
    )
    axs[1].set_xlabel("Count")
    xs = np.linspace(-np.pi / 2, np.pi / 2, 111)
    axs[1].plot(np.cos(xs) / 2, xs)

    # Compute the integral and the uncertainty,
    # assuming the samples are uncorrelated.
    chain = np.array(chain)
    eint = (chain**2).mean()
    uint = (chain**2).std() / np.sqrt(chain.size)
    print(f"integral: {eint:.4f} ± {uint:.4f}")


demo_cosine()
```

```
integral: 0.4609 ± 0.0049
```



A few remarks:

- When the number of samples is reduced to 1000, the quality of the histogram becomes poor. The cause for this problem is shown in the left plot: successive samples are *NOT* independent, meaning that the actual information content is much lower than the number of samples.
- In addition, the error estimate is clearly too optimistic. It is computed with the assumption that the samples are independent, which is clearly not the case. A correct error estimate should take into account the covariance of successive steps, which goes beyond the scope of this notebook.

The Cannonball game with a timer

The following plot illustrates a variation on a popular computer game, "Cannonball". The game proceeds as follows:

1. Castle A fires an explosive with a timer at castle B. Castle A can control the initial velocity vector of the explosive.
2. Five seconds after launch, the explosive detonates.
3. If the explosive reaches the roof of castle B after five seconds, castle A wins.
4. The usual war rhetoric: If we do not destroy B, B will destroy us. We only have one chance to win.

Note: all numerical values in this game are in SI base units.

```
def configure_game(seed=1):
    """Generate a random terrain and castle positions.

    Parameters
    ----------
    seed
        Seed for the random number generator.

    Returns
    -------
    (xs, ys)
        Coordinates describing the terrain, used for visual only.
    castle_a
        (x, y) coordinates for the position of Castle A.
    castle_b
        (x, y) coordinates for the position of Castle B.

    """
    rng = np.random.default_rng(seed)

    # Compute the terrain with a stochastic process.
    nterrain = 201
    xs = np.linspace(-100, 100, nterrain)
    ys = np.zeros(nterrain)
    for i in range(0, nterrain - 1):
        ys[i + 1] = ys[i] * 0.99 + rng.normal(0, 2)

    # The positions of the two castles
    ia = rng.integers(40, 60)
    castle_a = xs[ia], ys[ia] - 3
    ib = rng.integers(140, 160)
    castle_b = xs[ib], ys[ib] - 3

    return (xs, ys), castle_a, castle_b


def compute_trajectory(pos0, vel0):
    """Compute the 5-second trajectory of the explosive.

    The trajectory takes into account the gravitational
    force, friction of the explosive with the air and
    the wind.

    Parameters
```

(continues on next page)

(continued from previous page)

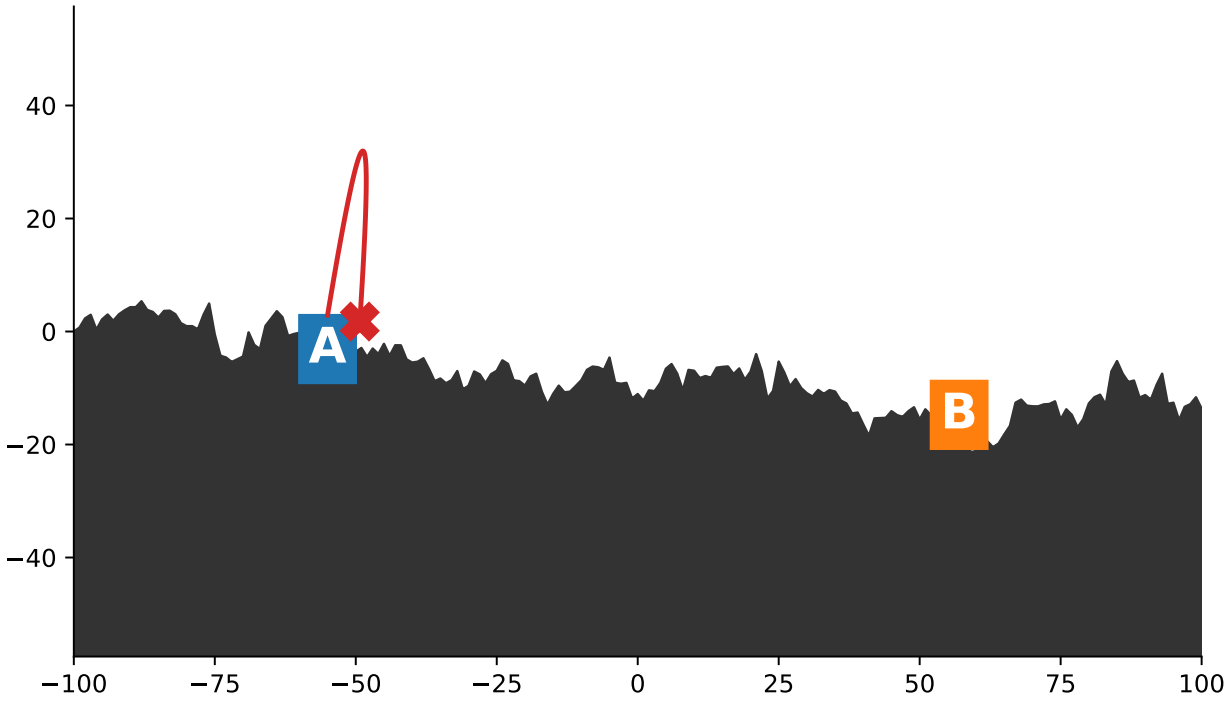
```
    ----------
    pos0
        The initial position vector,
        the middle of the roof of a Castle.
    vel0
        The initial velocity vector.

    Returns
    -------
    trajectory
        The trajectory of the explosive as a 2D array.
        First row contains x-positions, second-row y-positions.
        Columns correspond to time steps.

    """
    STD_GRAVITY = 9.81
    FRICTION_COEFF = 0.3
    WIND_VEL = -3.0
    TIMER = 5.0

    def odefun(time, velpos):
        vel, pos = np.split(velpos, 2)
        acc = np.array(
            [
                -FRICTION_COEFF * (vel[0] - WIND_VEL),
                -STD_GRAVITY - FRICTION_COEFF * vel[1],
            ]
        )
        return np.concatenate([acc, vel])

    velpos0 = np.concatenate([vel0, pos0])
    sol = solve_ivp(
        odefun,
        [0, TIMER],
        velpos0,
        t_eval=np.linspace(0, TIMER, 101),
        rtol=1e-9,
        atol=1e-9,
    )
    return np.split(sol.y, 2)[1]


def plot_game(terrain, castle_a, castle_b, trajectory, num):
    """Plot the outcome of a game.

    Parameters
    ----------
    terrain, castle_a, castle_b
        Return values of the configure_game function.
    trajectory
        Return value of the compute_trajectory function.

    """
```

(continues on next page)

(continued from previous page)

```
    CASTLE_WIDTH = 10
    CASTLE_HEIGHT = 12

    plt.close(num)
    fig, ax = plt.subplots(figsize=(7, 4), num=num)
    ax.fill_between(terrain[0], terrain[1], -100, color="#333333")
    xya = (castle_a[0] - CASTLE_WIDTH / 2, castle_a[1])
    ax.add_patch(Rectangle(xya, CASTLE_WIDTH, CASTLE_HEIGHT, color="C0"))
    ax.text(
        castle_a[0],
        castle_a[1] + CASTLE_HEIGHT / 2,
        "A",
        color="w",
        va="center",
        ha="center",
        fontsize=20,
        weight="bold",
    )
    xyb = (castle_b[0] - CASTLE_WIDTH / 2, castle_b[1])
    ax.add_patch(Rectangle(xyb, CASTLE_WIDTH, CASTLE_HEIGHT, color="C1"))
    ax.text(
        castle_b[0],
        castle_b[1] + CASTLE_HEIGHT / 2,
        "B",
        color="w",
        va="center",
        ha="center",
        fontsize=20,
        weight="bold",
    )
    ax.plot(trajectory[0], trajectory[1], color="C3", lw=2)
    ax.plot(trajectory[0][-1], trajectory[1][-1], "C3X", ms=15)
    ax.set_aspect("equal", adjustable="datalim")
    ax.set_xlim(-100, 100)
    ax.set_ylim(-50, 50)


def demo_game_init(x_vel, y_vel):
    """Example game with fixed initial velocity vector.

    In this example, castle A makes a catastrophic mistake.
    """

    CASTLE_HEIGHT = 12

    terrain, castle_a, castle_b = configure_game()
    pos0_a = np.array([castle_a[0], castle_a[1] + CASTLE_HEIGHT])
    vel0_a = np.array([x_vel, y_vel])
    trajectory = compute_trajectory(pos0_a, vel0_a)
    plot_game(terrain, castle_a, castle_b, trajectory, "game_traj_init")
```

```
demo_game_init(5, 30)
```



To win the game, castle A must find the correct initial velocity vector. This problem is solved below by finding a distribution of initial velocities that will result in a desired distribution of final positions of the explosive.

- The desired distribution for the final position is normally distributed at the center of the roof of castle B with a standard deviation of 1 meter.
- The corresponding distribution of velocities has no closed analytical form. This distribution is of interest, not only because the mean is a good choice of the initial velocity. The spread of the distribution also tells us how precisely the initial velocity must be specified.

The code below does the following:

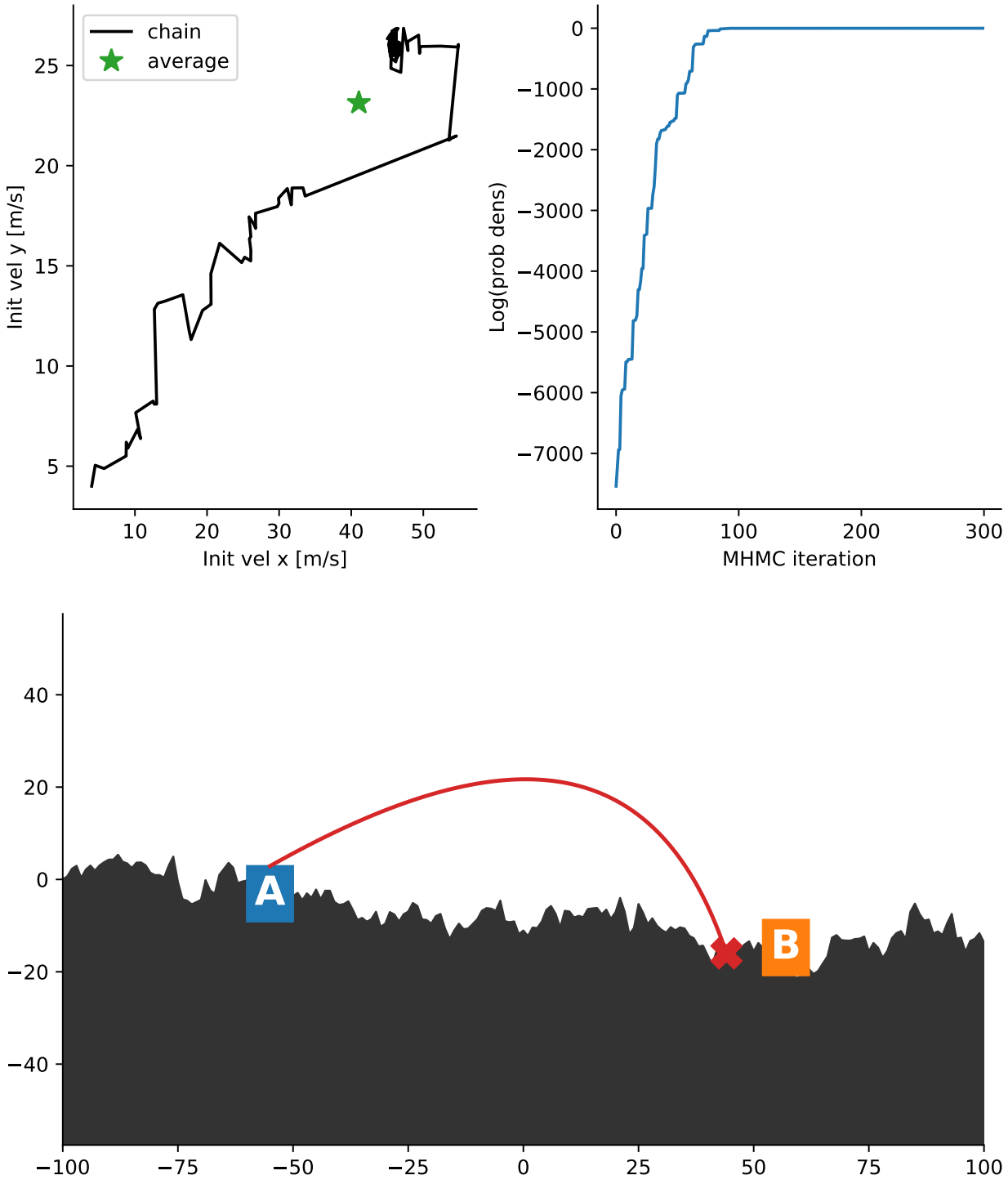
- Define MHMC sampling and collect 300 sample points
- Visualize the initial velocity distribution.
- Plot the trajectory using the average of all initial velocities.

```
def demo_game_velocities():
    """Demonstration of MHMC chain to win canon ball in one move."""

    CASTLE_HEIGHT = 12

    terrain, castle_a, castle_b = configure_game()
    pos0_a = np.array([castle_a[0], castle_a[1] + CASTLE_HEIGHT])
    rng = np.random.default_rng()

    def ln_stat_dens(vel0_a):
        """Compute the probability density at the final positions."""
        trajectory = compute_trajectory(pos0_a, vel0_a)
        pos1_b = trajectory[:, -1]
        pos0_b = np.array([castle_b[0], castle_b[1] + CASTLE_HEIGHT])
        sigma = 1.0
        delta = pos1_b - pos0_b
        return -np.dot(delta, delta) / (2 * sigma)
```

(continues on next page)

(continued from previous page)

```
    def prop_gen(xcur):
        """Propose a change in initial velocities."""
        # return xcur + rng.standard_normal(size=2) * 2
        return xcur + rng.standard_cauchy(size=2) / 2

    vel0_a_init = np.array([4.0, 4.0])
    chain, ln_sds = mhmc_driver(300, vel0_a_init, ln_stat_dens, prop_gen)
    chain = np.array(chain)

    # Discard the first steps (burn-in)
    ndiscard = 0
    chain = chain[ndiscard:]
    ln_sds = ln_sds[ndiscard:]

    # Plot the MHMC results
    plt.close("game_mhmc")
    fig, axs = plt.subplots(1, 2, figsize=(7, 4), num="game_mhmc")
    vel0_a_mean = chain.mean(axis=0)
    axs[0].plot(chain[:, 0], chain[:, 1], "k-", label="chain")
    axs[0].plot(vel0_a_mean[0], vel0_a_mean[1], "C2*", ms=12, label="average")
    axs[0].set_xlabel("Init vel x [m/s]")
    axs[0].set_ylabel("Init vel y [m/s]")
    axs[0].legend(loc=0)
    axs[1].plot(ln_sds)
    axs[1].set_xlabel("MHMC iteration")
    axs[1].set_ylabel("Log(prob dens)")

    # Plot the game outcome for the last state in the chain
    trajectory = compute_trajectory(pos0_a, vel0_a_mean)
    plot_game(terrain, castle_a, castle_b, trajectory, num="game_traj_opt")


demo_game_velocities()
```



Remarks:

- The Markov chain exhibits a significant amount of **burn-in**. The first state of the initial velocity vector is so far out of distribution, that several iterations are needed to find the center of the distribution. For such a short run, the burn-in phase biases the averages. In the limit of very long MC chains, burn-in becomes negligible. However, such long runs are usually too costly and one resorts to manual removal of the burn-in. This is often a subjective choice, for which automatic methods have recently been proposed.
- **The Cauchy distribution is an interesting proposal distribution.** Due to the heavy tails of the distribution, both large steps (exploration) and small steps (refinement) are considered in one chain. The same strategy is used

by wild animals to optimize their efficiency when gathering food, which is known as the Lévy Flight Foraging Hypothesis. Try replacing the Cauchy distribution with a normal distribution: A large standard deviation is needed to quickly overcome the burn-in, but then the steps are generally too large for an efficient sampling of the sharply peaked distribution.

- Instead of using a Cauchy distribution, one can also work with **adaptive step sizes** in the proposal distribution. Naive approaches to control the step size, based on the acceptance rate of previous iterations, may bias the stationary distribution. Use these with care. Specialized algorithms have been developed to deal specifically with this difficulty, e.g. the Affine Invariant Markov Chain implemented in emcee is a good solution for when step sizes are difficult to set manually.
- **This game is a simplified model for many problems in physics** with the same statistical structure:
  - *"For what distribution of physical parameters is a particular process or outcome observed?"*

    > For such problems, the parameters can be found by coupling an MHMC algorithm to a computer-controlled experimental setup. This approach is also used to discover new materials, to crystallize proteins, to discover new pharmaceuticals, etc.
    >
    > Other algorithms than MHMC, such as genetic algorithms or the particle swarm method, are popular for this purpose. Just remember that they don't have a stationary distribution, unless they are cast into an MHMC framework.

  - *"What is the distribution of model parameters that can explain a distribution of measurements?"*

    > This type of modeling is called Bayesian inference and goes beyond the scope of this section.

- One **important class of simulations is not covered in this section**, namely simulations of physical systems characterized by a probability density. These will be treated in the Statistical Physics course. Needless to say, MHMC is one of the main simulation workhorses in statistical physics. Besides MHMC, one can also use (stochastic or chaotic) numerical integrators to sample probability densities. All these techniques are beyond the scope of this section.

Extended Cannonball game

Consider the following **extension of the above game**: treat the wind speed as an extra stochastic quantity. Assume that it has a normal distribution with a mean of 0 m/s and a standard deviation of 5 m/s. You can consider two scenarios:

1. Castle A has no control over the wind speed, but can measure it. Find out how to adjust the distribution of initial explosive velocities as a function of the wind speed. Instead of sweeping through all wind speeds, treat it as an extra stochastic quantity and collect all the relevant information in one MHMC chain.
2. Castle A has no control over the wind speed, and cannot measure it. What is the probability it will still win the game?

Both questions can be solved by analyzing the same MHMC chain.