# Tutorial 10

## Scientific algorithmization in practice

Jan Bures

18YZALG – Basics of Algorithmization, Summer Semester 2026

# How today works

- We will not prove numerical analysis theorems.
- We will turn formulas into small Python algorithms and ask: **how accurate, how fast, and when does it fail?**
- Every topic has a matching notebook and a small mini-project idea.

**Practical rule**

Scientific algorithmization is mostly about choosing a computable approximation and checking its error honestly.

# How today works

- We will not prove numerical analysis theorems.
- We will turn formulas into small Python algorithms and ask: **how accurate, how fast, and when does it fail?**
- Every topic has a matching notebook and a small mini-project idea.

**Practical rule**

Scientific algorithmization is mostly about choosing a computable approximation and checking its error honestly.

# How today works

- We will not prove numerical analysis theorems.
- We will turn formulas into small Python algorithms and ask: **how accurate, how fast, and when does it fail?**
- Every topic has a matching notebook and a small mini-project idea.

**Practical rule**

Scientific algorithmization is mostly about choosing a computable approximation and checking its error honestly.

# How today works

- We will not prove numerical analysis theorems.
- We will turn formulas into small Python algorithms and ask: **how accurate, how fast, and when does it fail?**
- Every topic has a matching notebook and a small mini-project idea.

**Practical rule**

Scientific algorithmization is mostly about choosing a computable approximation and checking its error honestly.

# How today works

- We will not prove numerical analysis theorems.
- We will turn formulas into small Python algorithms and ask: **how accurate, how fast, and when does it fail?**
- Every topic has a matching notebook and a small mini-project idea.

**Practical rule**

Scientific algorithmization is mostly about choosing a computable approximation and checking its error honestly.

# What makes this different from symbolic math?

**Symbolic question**

Can I derive an exact formula?

**Example**

Differentiate $f(x)$ by applying rules on paper.

**Algorithmic question**

Can I compute a useful numerical answer with controlled error?

**Example**

Estimate $f'(x)$ from values $f(x-h)$, $f(x)$ and $f(x+h)$.

**Today**

The output is not only a number. The output is also an explanation of accuracy, cost, and limitations.

# What makes this different from symbolic math?

**Symbolic question**

Can I derive an exact formula?

**Example**

Differentiate $f(x)$ by applying rules on paper.

**Algorithmic question**

Can I compute a useful numerical answer with controlled error?

**Example**

Estimate $f'(x)$ from values $f(x-h)$, $f(x)$ and $f(x+h)$.

**Today**

The output is not only a number. The output is also an explanation of accuracy, cost, and limitations.

# Three recurring questions

1. **Model:** what mathematical object are we computing with? A function, a table, a matrix, a dataset?
2. **Approximation:** which simplified computation do we use? Polynomial, line segments, finite difference, linear solve, projection?
3. **Check:** how do we know it worked? Error on known examples, convergence, residual, reconstruction error, runtime.

# Three recurring questions

1. **Model:** what mathematical object are we computing with? A function, a table, a matrix, a dataset?
2. **Approximation:** which simplified computation do we use? Polynomial, line segments, finite difference, linear solve, projection?
3. **Check:** how do we know it worked? Error on known examples, convergence, residual, reconstruction error, runtime.

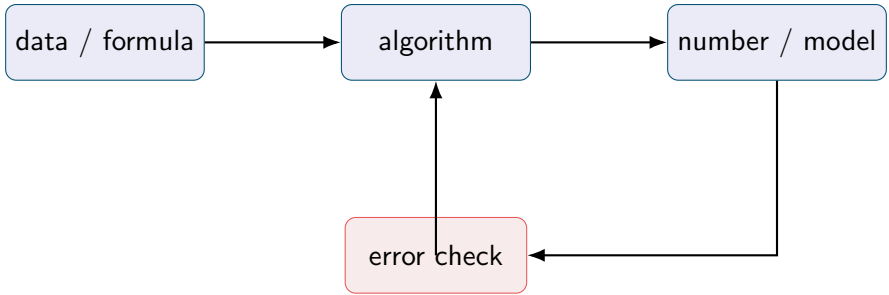
# Three recurring questions

1. **Model:** what mathematical object are we computing with? A function, a table, a matrix, a dataset?

2. **Approximation:** which simplified computation do we use? Polynomial, line segments, finite difference, linear solve, projection?

3. **Check:** how do we know it worked? Error on known examples, convergence, residual, reconstruction error, runtime.

# The shared computational pattern




**Practical mindset**

Numerical methods are usually iterative improvements: compute, measure error, adjust parameters.

# 1. Taylor expansion: why do we care?

**Problem**

Some functions are expensive, unavailable, or only needed near one point.

**Algorithmic idea**

Replace the function by a polynomial that matches its value and derivatives near a chosen point.

**Example around 0**

$$e^x \approx 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

# 1. Taylor expansion: why do we care?

**Problem**

Some functions are expensive, unavailable, or only needed near one point.

**Algorithmic idea**

Replace the function by a polynomial that matches its value and derivatives near a chosen point.

**Example around 0**

$$e^x \approx 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

# 1. Taylor expansion: why do we care?

**Problem**

Some functions are expensive, unavailable, or only needed near one point.

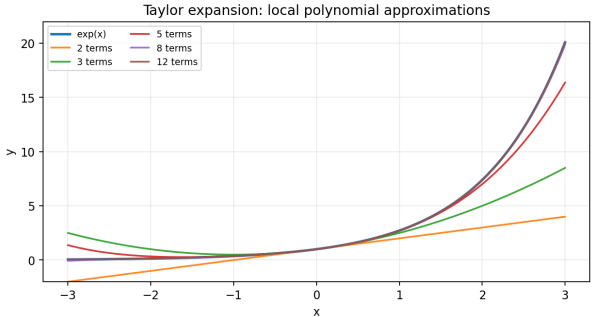
**Algorithmic idea**

Replace the function by a polynomial that matches its value and derivatives near a chosen point.

**Example around 0**

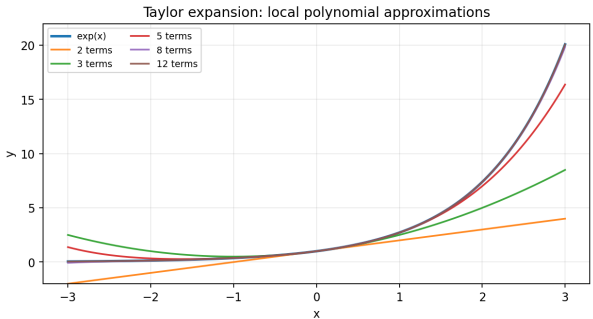
$$e^x \approx 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

# Taylor expansion is local



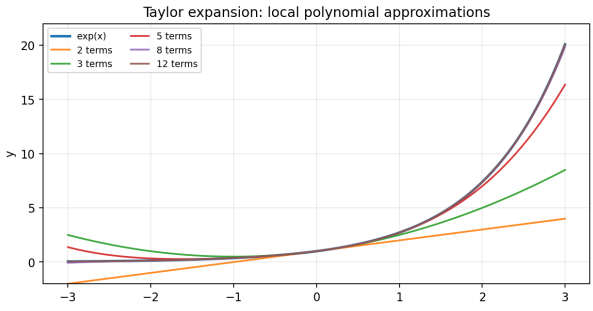

- Near the expansion point, few terms may be enough.
- Farther away, the same polynomial can become poor.
- Adding terms improves accuracy, but costs more work.

# Taylor expansion is local




- Near the expansion point, few terms may be enough.
- Farther away, the same polynomial can become poor.
- Adding terms improves accuracy, but costs more work.

# Taylor expansion is local




- Near the expansion point, few terms may be enough.
- Farther away, the same polynomial can become poor.
- Adding terms improves accuracy, but costs more work.

# Taylor expansion: simple implementation

## Code pattern

```
def taylor_exp(x, n_terms):
    total = 0.0
    power = 1.0
    factorial = 1.0
    for k in range(n_terms):
        if k > 0:
            power *= x
            factorial *= k
        total += power / factorial
    return total
```

## Cost

With this direct loop, evaluating $n$ terms is $\mathcal{O}(n)$ time and $\mathcal{O}(1)$ extra memory.

# Taylor expansion: simple implementation

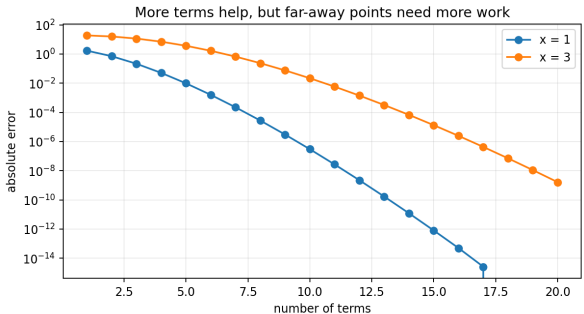
## Code pattern

```
def taylor_exp(x, n_terms):
    total = 0.0
    power = 1.0
    factorial = 1.0
    for k in range(n_terms):
        if k > 0:
            power *= x
            factorial *= k
        total += power / factorial
    return total
```

## Cost

With this direct loop, evaluating $n$ terms is $\mathcal{O}(n)$ time and $\mathcal{O}(1)$ extra memory.

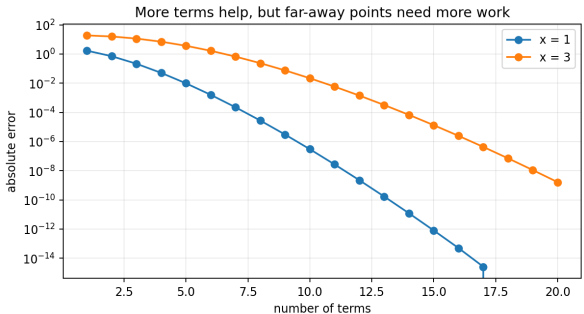
# Taylor error: more terms versus farther point




## What students should notice

- Error decreases as terms are added.
- $x = 3$ needs more terms than $x = 1$.
- "Enough terms" depends on the target tolerance.

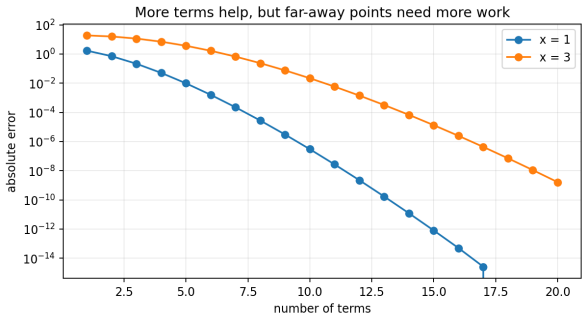
# Taylor error: more terms versus farther point




## What students should notice

- Error decreases as terms are added.
- $x = 3$ needs more terms than $x = 1$.
- "Enough terms" depends on the target tolerance.

# Taylor error: more terms versus farther point




## What students should notice

- Error decreases as terms are added.
- $x = 3$ needs more terms than $x = 1$.
- "Enough terms" depends on the target tolerance.

# Notebook demo 1: Taylor expansion

**Open**

`notebooks/01_taylor_expansion.ipynb`

- Show approximations of $e^x$ for 2, 3, 5, 8 and 12 terms.
- Print an error table for $x = 0.1, 1, 2, 4$.
- Run the tiny timing cell and connect it to $\mathcal{O}(n)$.

# Notebook demo 1: Taylor expansion

**Open**

`notebooks/01_taylor_expansion.ipynb`

- Show approximations of $e^x$ for 2, 3, 5, 8 and 12 terms.
- Print an error table for $x = 0.1, 1, 2, 4$.
- Run the tiny timing cell and connect it to $\mathcal{O}(n)$.

# Notebook demo 1: Taylor expansion

**Open**

`notebooks/01_taylor_expansion.ipynb`

- Show approximations of $e^x$ for 2, 3, 5, 8 and 12 terms.
- Print an error table for $x = 0.1, 1, 2, 4$.
- Run the tiny timing cell and connect it to $\mathcal{O}(n)$.

# Taylor: practical takeaways

| Use it when | Be careful when | Simple check |
|---|---|---|
| You need a local approximation near a known point. | You evaluate far from the expansion point. | Compare with a known function or add terms until change is small. |
| You want a fast polynomial approximation. | The function has singularities or sharp behavior nearby. | Plot error against number of terms. |

**Mini-project link**

Build a calculator that chooses the number of Taylor terms automatically for a requested tolerance.

# 2. Interpolation: values between samples

**Problem**

We measured a quantity at a few points, but we need an estimate at a point between them.

**Examples**

- Sensor calibration table.
- Weather measurements between hours.
- A curve sampled by a simulation.

# 2. Interpolation: values between samples

**Problem**

We measured a quantity at a few points, but we need an estimate at a point between them.

**Examples**

- Sensor calibration table.
- Weather measurements between hours.
- A curve sampled by a simulation.

# 2. Interpolation: values between samples

**Problem**

We measured a quantity at a few points, but we need an estimate at a point between them.

**Examples**

- Sensor calibration table.
- Weather measurements between hours.
- A curve sampled by a simulation.

# 2. Interpolation: values between samples

**Problem**

We measured a quantity at a few points, but we need an estimate at a point between them.

**Examples**

- Sensor calibration table.
- Weather measurements between hours.
- A curve sampled by a simulation.

# Piecewise linear interpolation

## Between two neighboring samples

If $x$ lies between $x_0$ and $x_1$, compute a fraction $t$ and blend:

$$t = \frac{x - x_0}{x_1 - x_0}, \qquad y \approx (1 - t)y_0 + ty_1.$$

## Code pattern

```
def lerp(x0, y0, x1, y1, x):
    t = (x - x0) / (x1 - x0)
    return (1 - t) * y0 + t * y1
```

# Piecewise linear interpolation

## Between two neighboring samples

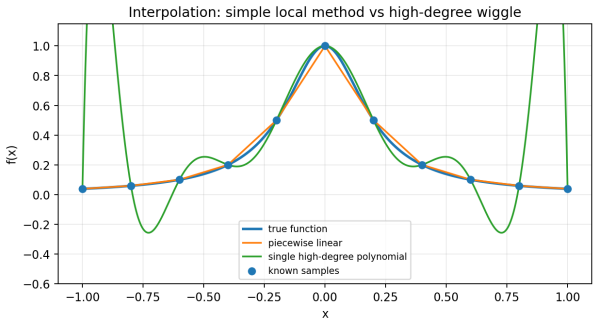
If $x$ lies between $x_0$ and $x_1$, compute a fraction $t$ and blend:

$$t = \frac{x - x_0}{x_1 - x_0}, \qquad y \approx (1 - t)y_0 + ty_1.$$

## Code pattern

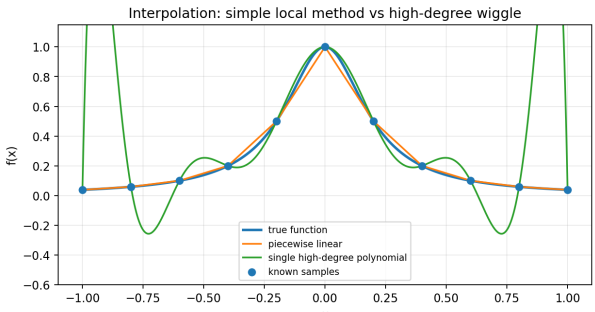
```
def lerp(x0, y0, x1, y1, x):
    t = (x - x0) / (x1 - x0)
    return (1 - t) * y0 + t * y1
```

# Interpolation: simple does not mean bad



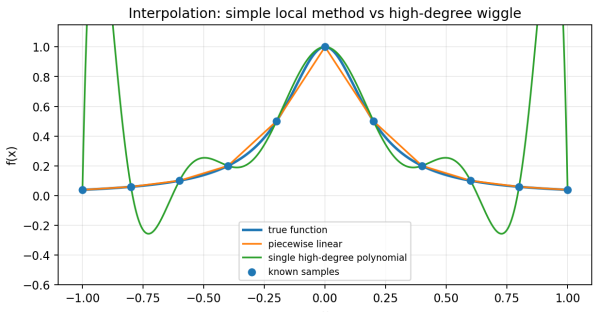

- Linear interpolation is local and predictable.
- A single high-degree polynomial can pass through all points and still behave badly between them.
- In practice, stable local methods are often better than impressive global formulas.

# Interpolation: simple does not mean bad




- Linear interpolation is local and predictable.
- A single high-degree polynomial can pass through all points and still behave badly between them.
- In practice, stable local methods are often better than impressive global formulas.

# Interpolation: simple does not mean bad




- Linear interpolation is local and predictable.
- A single high-degree polynomial can pass through all points and still behave badly between them.
- In practice, stable local methods are often better than impressive global formulas.

# Manual search for the interval

## Python idea

```
import numpy as np

def linear_interpolate(xs, ys, xq):
    i = np.searchsorted(xs, xq) - 1
    i = max(0, min(i, len(xs) - 2))
    x0, x1 = xs[i], xs[i + 1]
    y0, y1 = ys[i], ys[i + 1]
    t = (xq - x0) / (x1 - x0)
    return (1 - t) * y0 + t * y1
```

## Cost

Binary search for the interval is $\mathcal{O}(\log n)$; the interpolation formula itself is $\mathcal{O}(1)$.

# Manual search for the interval

## Python idea

```
import numpy as np

def linear_interpolate(xs, ys, xq):
    i = np.searchsorted(xs, xq) - 1
    i = max(0, min(i, len(xs) - 2))
    x0, x1 = xs[i], xs[i + 1]
    y0, y1 = ys[i], ys[i + 1]
    t = (xq - x0) / (x1 - x0)
    return (1 - t) * y0 + t * y1
```

## Cost

Binary search for the interval is $\mathcal{O}(\log n)$; the interpolation formula itself is $\mathcal{O}(1)$.

# Notebook demo 2: Interpolation

**Open**

`notebooks/02_interpolation.ipynb`

- Estimate a resistance value between calibration measurements.
- Reimplement piecewise linear interpolation manually.
- Show why a high-degree polynomial may wiggle.

# Notebook demo 2: Interpolation

**Open**

`notebooks/02_interpolation.ipynb`

- Estimate a resistance value between calibration measurements.
- Reimplement piecewise linear interpolation manually.
- Show why a high-degree polynomial may wiggle.

# Notebook demo 2: Interpolation

**Open**

`notebooks/02_interpolation.ipynb`

- Estimate a resistance value between calibration measurements.
- Reimplement piecewise linear interpolation manually.
- Show why a high-degree polynomial may wiggle.

# Interpolation: practical takeaways

| Method | Strength | Weakness |
|---|---|---|
| Nearest neighbor | simplest possible | jumps suddenly |
| Piecewise linear | stable, explainable, local | not smooth at sample points |
| High-degree polynomial | matches all samples exactly | may oscillate and be numerically unstable |

## Mini-project link

Create a small sensor-table estimator and compare nearest neighbor, linear interpolation and polynomial fitting.

# 3. Numerical differentiation: slope from values

**Problem**

We do not have a symbolic derivative, but we can evaluate or measure $f(x)$.

**Forward difference**

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

**Central difference**

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

# 3. Numerical differentiation: slope from values

**Problem**

We do not have a symbolic derivative, but we can evaluate or measure $f(x)$.

**Forward difference**

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

**Central difference**

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

# Derivative estimates in code

## Code pattern

```
def forward_difference(f, x, h):
    return (f(x + h) - f(x)) / h

def central_difference(f, x, h):
    return (f(x + h) - f(x - h)) / (2 * h)
```

## Important warning

Smaller $h$ is not always better. Very small $h$ subtracts nearly equal floating-point numbers.

# Derivative estimates in code

## Code pattern

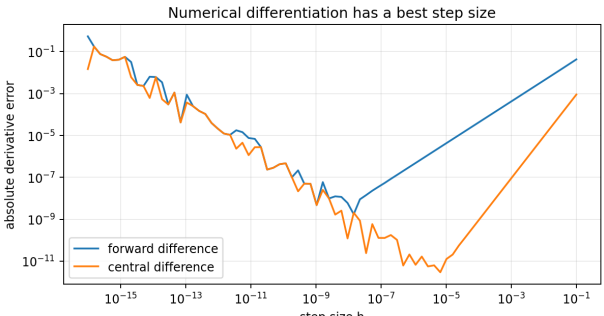
```
def forward_difference(f, x, h):
    return (f(x + h) - f(x)) / h

def central_difference(f, x, h):
    return (f(x + h) - f(x - h)) / (2 * h)
```

## Important warning

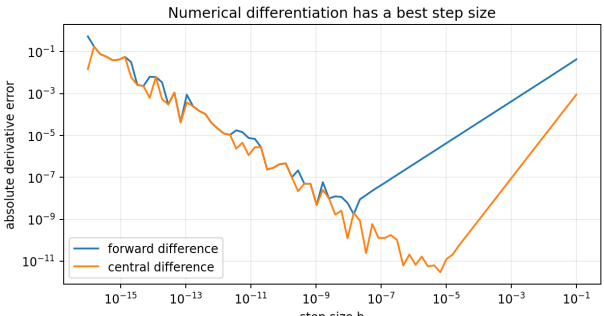
Smaller $h$ is not always better. Very small $h$ subtracts nearly equal floating-point numbers.

# Derivative error: there is a best step size



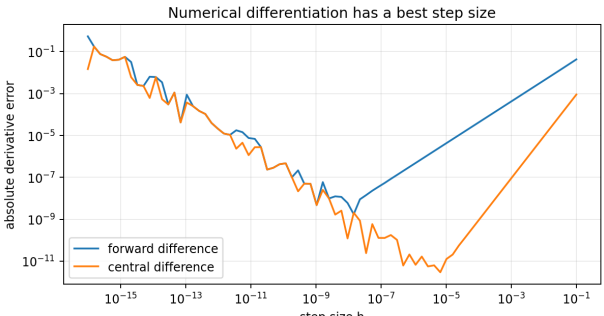

- Large $h$: approximation is too rough.
- Tiny $h$: round-off error dominates.
- Central difference usually improves accuracy for the same $h$.

# Derivative error: there is a best step size




- Large $h$: approximation is too rough.
- Tiny $h$: round-off error dominates.
- Central difference usually improves accuracy for the same $h$.

# Derivative error: there is a best step size




- Large $h$: approximation is too rough.
- Tiny $h$: round-off error dominates.
- Central difference usually improves accuracy for the same $h$.

# Numerical integration: area from samples

**Problem**

Estimate the area under a curve when exact integration is inconvenient.

**Trapezoid rule**

Split interval into small pieces and approximate each piece by a trapezoid.

**Simpson rule**

Use parabolic pieces; often more accurate for smooth functions.

# Numerical integration: area from samples

**Problem**

Estimate the area under a curve when exact integration is inconvenient.

**Trapezoid rule**

Split interval into small pieces and approximate each piece by a trapezoid.

**Simpson rule**

Use parabolic pieces; often more accurate for smooth functions.

# Trapezoid rule in code

## Code pattern

```
def trapezoid(f, a, b, n):
    xs = np.linspace(a, b, n + 1)
    ys = f(xs)
    h = (b - a) / n
    return h * (0.5*ys[0] + ys[1:-1].sum() + 0.5*ys[-1])
```

## Cost

Most simple quadrature rules are $\mathcal{O}(n)$: the main work is evaluating the function at $n$ or $n+1$ points.

# Trapezoid rule in code

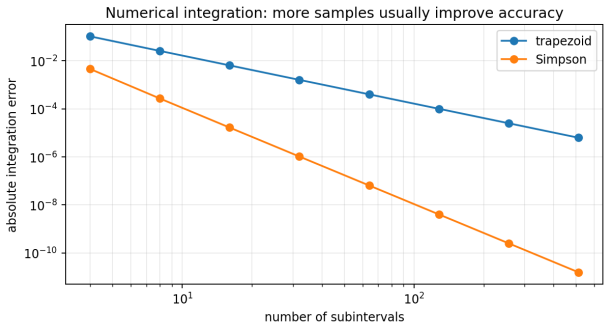
## Code pattern

```
def trapezoid(f, a, b, n):
    xs = np.linspace(a, b, n + 1)
    ys = f(xs)
    h = (b - a) / n
    return h * (0.5*ys[0] + ys[1:-1].sum() + 0.5*ys[-1])
```

## Cost

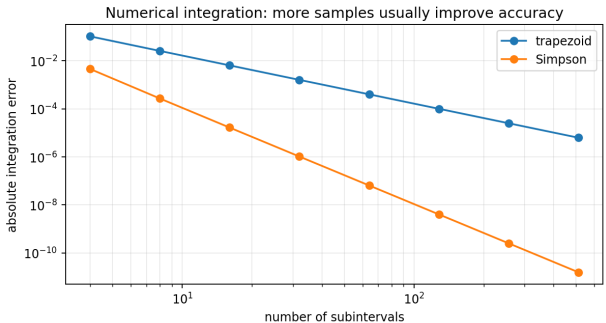
Most simple quadrature rules are $\mathcal{O}(n)$: the main work is evaluating the function at $n$ or $n+1$ points.

# Integration convergence



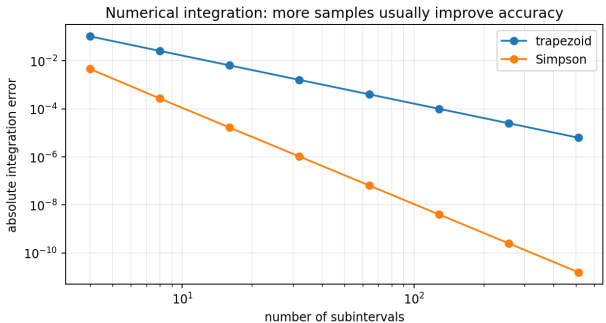

- More subintervals usually reduce error.
- Simpson can be much more accurate for smooth functions.
- A convergence plot is more informative than one lucky number.

# Integration convergence




- More subintervals usually reduce error.
- Simpson can be much more accurate for smooth functions.
- A convergence plot is more informative than one lucky number.

# Integration convergence




- More subintervals usually reduce error.
- Simpson can be much more accurate for smooth functions.
- A convergence plot is more informative than one lucky number.

# Notebook demo 3: differentiation and integration

**Open**

`notebooks/03_numerical_differentiation_integration.ipynb`

- Compare forward and central differences for $\sin(x)$.
- Plot derivative error against $h$.
- Integrate $\sin(x)$ from 0 to $\pi$ and compare with exact area 2.

# Notebook demo 3: differentiation and integration

**Open**

`notebooks/03_numerical_differentiation_integration.ipynb`

- Compare forward and central differences for $\sin(x)$.
- Plot derivative error against $h$.
- Integrate $\sin(x)$ from 0 to $\pi$ and compare with exact area 2.

# Notebook demo 3: differentiation and integration

**Open**

`notebooks/03_numerical_differentiation_integration.ipynb`

- Compare forward and central differences for $\sin(x)$.
- Plot derivative error against $h$.
- Integrate $\sin(x)$ from 0 to $\pi$ and compare with exact area 2.

# Differentiation and integration: takeaways

| Task | Main parameter | What to show in a demo |
|---|---|---|
| Differentiate | step size $h$ | error vs $h$; central vs forward |
| Integrate | number of subintervals $n$ | error vs $n$; trapezoid vs Simpson |

**Mini-project link**

Estimate speed and travelled distance from a sampled position signal.

# 4. Numerical linear algebra: why it appears everywhere

- Solving balance equations: $Ax = b$.
- Fitting a model to noisy data: least squares.
- Working with many variables at once: matrices and vectors.
- PCA, simulations and optimization all depend on linear algebra routines.

# 4. Numerical linear algebra: why it appears everywhere

- Solving balance equations: $Ax = b$.
- Fitting a model to noisy data: least squares.
- Working with many variables at once: matrices and vectors.
- PCA, simulations and optimization all depend on linear algebra routines.

# 4. Numerical linear algebra: why it appears everywhere

- Solving balance equations: $Ax = b$.
- Fitting a model to noisy data: least squares.
- Working with many variables at once: matrices and vectors.
- PCA, simulations and optimization all depend on linear algebra routines.

# 4. Numerical linear algebra: why it appears everywhere

- Solving balance equations: $Ax = b$.
- Fitting a model to noisy data: least squares.
- Working with many variables at once: matrices and vectors.
- PCA, simulations and optimization all depend on linear algebra routines.

# Solve systems directly, do not invert explicitly

## Recommended pattern

```
import numpy as np

x = np.linalg.solve(A, b)        # good: solve A x = b
# x = np.linalg.inv(A) @ b       # avoid in normal use
```

## Reason

Computing the inverse usually does extra work and can be less stable. Ask for the quantity you actually need: the solution $x$.

# Solve systems directly, do not invert explicitly

## Recommended pattern

```
import numpy as np

x = np.linalg.solve(A, b)        # good: solve A x = b
# x = np.linalg.inv(A) @ b       # avoid in normal use
```

## Reason

Computing the inverse usually does extra work and can be less stable. Ask for the quantity you actually need: the solution $x$.

# A residual check is simple and powerful

**After solving $Ax = b$**

Compute the residual

$$r = Ax - b.$$

**Interpretation**

If $\|r\|$ is small, your computed $x$ satisfies the equations approximately.

**But**

A tiny residual does not always mean the original problem is well-conditioned.

# A residual check is simple and powerful

**After solving $Ax = b$**

Compute the residual

$$r = Ax - b.$$

**Interpretation**

If $\|r\|$ is small, your computed $x$ satisfies the equations approximately.

**But**

A tiny residual does not always mean the original problem is well-conditioned.

# A residual check is simple and powerful

**After solving $Ax = b$**

Compute the residual

$$r = Ax - b.$$

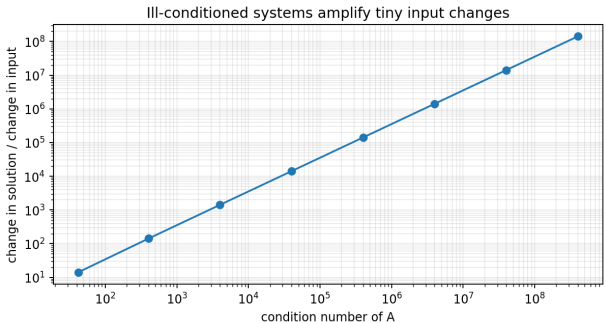
**Interpretation**

If $\|r\|$ is small, your computed $x$ satisfies the equations approximately.

**But**

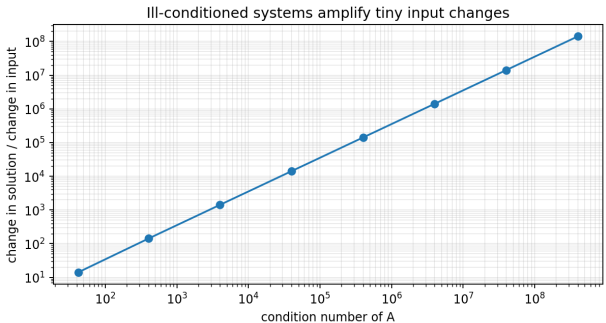
A tiny residual does not always mean the original problem is well-conditioned.

# Conditioning: when tiny changes become big changes



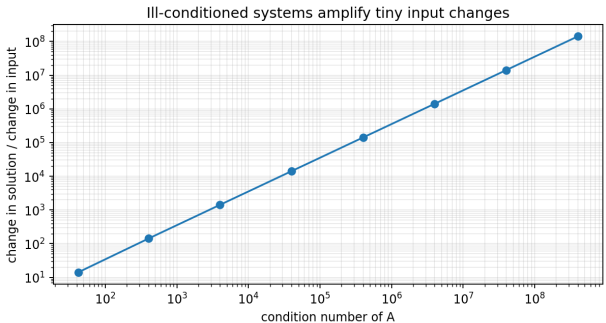

- Some matrices are nearly singular.
- Then small measurement noise in $b$ can strongly change $x$.
- This is not a Python bug; it is a property of the mathematical problem.

# Conditioning: when tiny changes become big changes



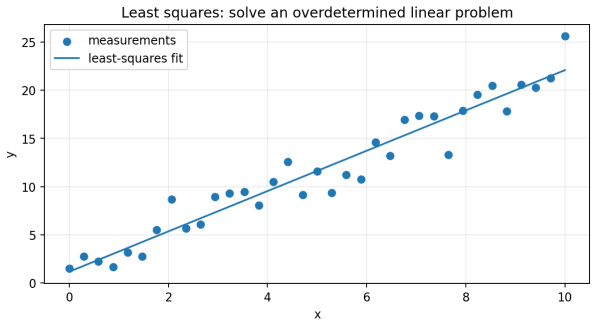

- Some matrices are nearly singular.
- Then small measurement noise in $b$ can strongly change $x$.
- This is not a Python bug; it is a property of the mathematical problem.

# Conditioning: when tiny changes become big changes




- Some matrices are nearly singular.
- Then small measurement noise in $b$ can strongly change $x$.
- This is not a Python bug; it is a property of the mathematical problem.

# Least squares: fitting noisy data



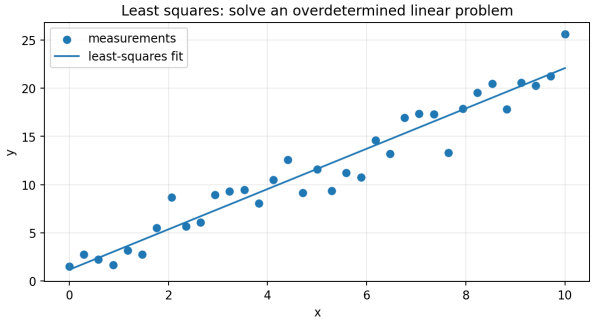

**Idea**

When there are more equations than unknowns, solve the best approximate system:

$$\min_x \|Ax - b\|_2.$$

- Fits a line, polynomial, or small model.
- The residual becomes the quality score.

# Least squares: fitting noisy data




**Idea**

When there are more equations than unknowns, solve the best approximate system:

$$\min_x \|Ax - b\|_2.$$

- Fits a line, polynomial, or small model.
- The residual becomes the quality score.

# Notebook demo 4: numerical linear algebra

**Open**

`notebooks/04_numerical_linear_algebra.ipynb`

- Solve a small $3 \times 3$ linear system and verify $Ax \approx b$.
- Compare `solve` with explicit inverse on small matrices.
- Show condition number and a least-squares line fit.

# Notebook demo 4: numerical linear algebra

**Open**

`notebooks/04_numerical_linear_algebra.ipynb`

- Solve a small $3 \times 3$ linear system and verify $Ax \approx b$.
- Compare `solve` with explicit inverse on small matrices.
- Show condition number and a least-squares line fit.

# Notebook demo 4: numerical linear algebra

**Open**

`notebooks/04_numerical_linear_algebra.ipynb`

- Solve a small $3 \times 3$ linear system and verify $Ax \approx b$.
- Compare `solve` with explicit inverse on small matrices.
- Show condition number and a least-squares line fit.

# Numerical linear algebra: practical takeaways

| Situation | Use | Check |
|---|---|---|
| Exact square system | `np.linalg.solve(A,b)` | residual $\|Ax - b\|$ |
| Noisy overdetermined data | `np.linalg.lstsq(A,b)` | residual / plot fit |
| Suspicious sensitivity | condition number | perturb input and observe output |

**Mini-project link**

Fit a simple model to noisy measurements and explain residuals, not only coefficients.

# 5. PCA: reduce dimensions while preserving variation

**Problem**

A dataset may have many features. We want to view or compress it using fewer coordinates.

**PCA idea**

Find directions in the data with the largest variance, then project data onto the first few directions.

**Why PCA comes after linear algebra**

PCA is a matrix algorithm: centering, covariance, eigenvectors or SVD, projection.

# 5. PCA: reduce dimensions while preserving variation

**Problem**

A dataset may have many features. We want to view or compress it using fewer coordinates.

**PCA idea**

Find directions in the data with the largest variance, then project data onto the first few directions.

**Why PCA comes after linear algebra**

PCA is a matrix algorithm: centering, covariance, eigenvectors or SVD, projection.

# 5. PCA: reduce dimensions while preserving variation

**Problem**

A dataset may have many features. We want to view or compress it using fewer coordinates.

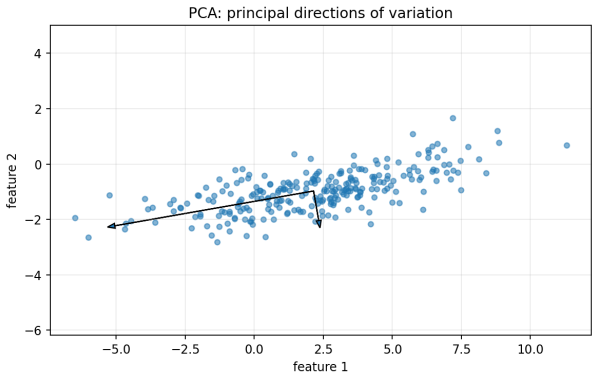
**PCA idea**

Find directions in the data with the largest variance, then project data onto the first few directions.

**Why PCA comes after linear algebra**

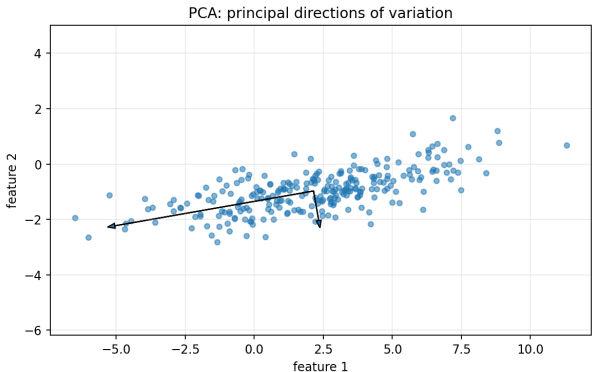
PCA is a matrix algorithm: centering, covariance, eigenvectors or SVD, projection.

# PCA in one picture



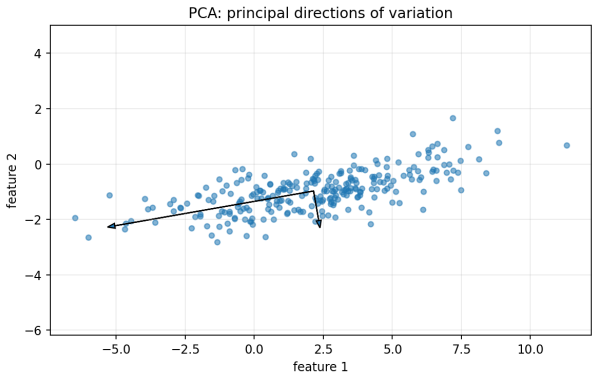

- Each point is one observation.
- Each axis is one feature.
- PC1 is the direction with most variation.
- PC2 is the next best perpendicular direction.

# PCA in one picture



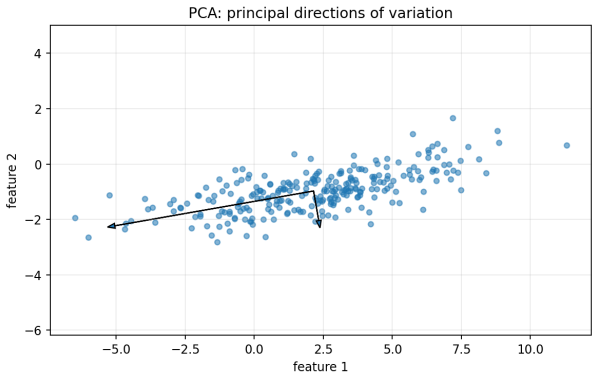

- Each point is one observation.
- Each axis is one feature.
- PC1 is the direction with most variation.
- PC2 is the next best perpendicular direction.

# PCA in one picture




- Each point is one observation.
- Each axis is one feature.
- PC1 is the direction with most variation.
- PC2 is the next best perpendicular direction.

# PCA in one picture




- Each point is one observation.
- Each axis is one feature.
- PC1 is the direction with most variation.
- PC2 is the next best perpendicular direction.

# PCA recipe in Python

## Core steps

```
X_centered = X - X.mean(axis=0)
C = np.cov(X_centered, rowvar=False)
values, vectors = np.linalg.eigh(C)
order = values.argsort()[::-1]
vectors = vectors[:, order]
Z = X_centered @ vectors[:, :k]   # k-dimensional representation
```

## Practical detail

Always center the data first. Otherwise the first component may describe the mean offset, not the shape of variation.

# PCA recipe in Python

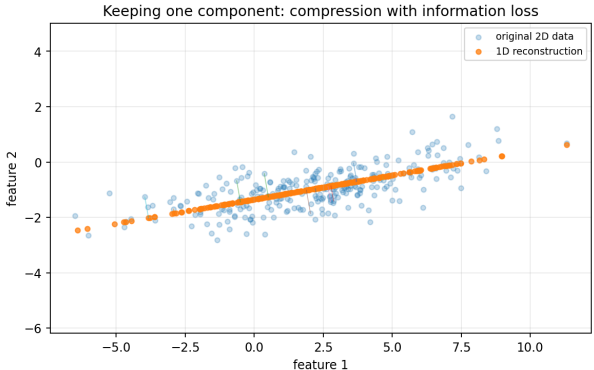
## Core steps

```
X_centered = X - X.mean(axis=0)
C = np.cov(X_centered, rowvar=False)
values, vectors = np.linalg.eigh(C)
order = values.argsort()[::-1]
vectors = vectors[:, order]
Z = X_centered @ vectors[:, :k]   # k-dimensional representation
```

## Practical detail

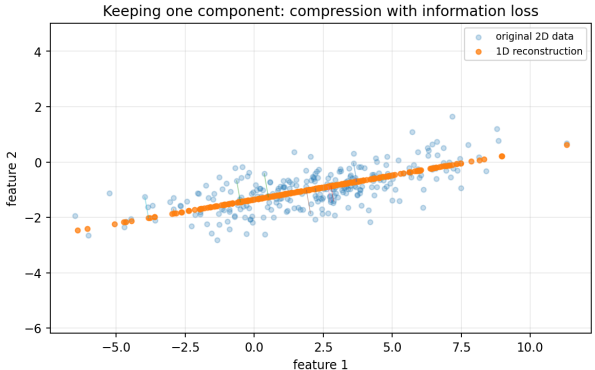
Always center the data first. Otherwise the first component may describe the mean offset, not the shape of variation.

# PCA compression and reconstruction



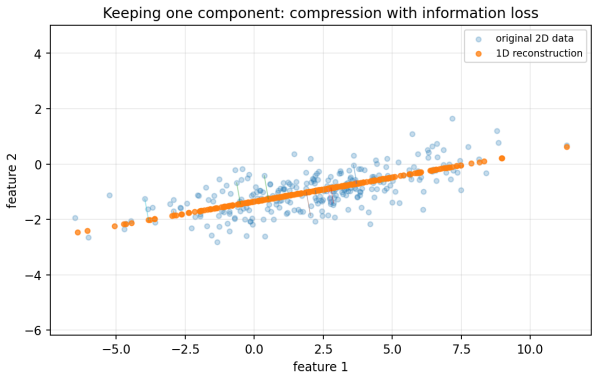

- Keeping one component turns each 2D point into one number.
- Reconstructing back to 2D loses information.
- Reconstruction error tells us how much detail was lost.

# PCA compression and reconstruction




- Keeping one component turns each 2D point into one number.
- Reconstructing back to 2D loses information.
- Reconstruction error tells us how much detail was lost.

# PCA compression and reconstruction




- Keeping one component turns each 2D point into one number.
- Reconstructing back to 2D loses information.
- Reconstruction error tells us how much detail was lost.

# Explained variance

## Meaning

Explained variance ratio answers: how much of the total variability is captured by each component?

## Simple interpretation

If PC1 explains 90 percent, a 1D projection may already be a good summary. If PC1 explains 40 percent, one component is probably too aggressive.

## Important warning

PCA finds linear directions. It does not understand labels, classes, or nonlinear structure by itself.

# Explained variance

## Meaning

Explained variance ratio answers: how much of the total variability is captured by each component?

## Simple interpretation

If PC1 explains 90 percent, a 1D projection may already be a good summary. If PC1 explains 40 percent, one component is probably too aggressive.

## Important warning

PCA finds linear directions. It does not understand labels, classes, or nonlinear structure by itself.

# Explained variance

## Meaning

Explained variance ratio answers: how much of the total variability is captured by each component?

## Simple interpretation

If PC1 explains 90 percent, a 1D projection may already be a good summary. If PC1 explains 40 percent, one component is probably too aggressive.

## Important warning

PCA finds linear directions. It does not understand labels, classes, or nonlinear structure by itself.

# Notebook demo 5: PCA

**Open**

`notebooks/05_pca.ipynb`

- Generate a correlated 2D dataset.
- Center the data and compute principal directions.
- Project to one component, reconstruct and measure error.
- Show the SVD version as the library-friendly implementation idea.

# Notebook demo 5: PCA

**Open**

`notebooks/05_pca.ipynb`

- Generate a correlated 2D dataset.
- Center the data and compute principal directions.
- Project to one component, reconstruct and measure error.
- Show the SVD version as the library-friendly implementation idea.

# Notebook demo 5: PCA

**Open**

`notebooks/05_pca.ipynb`

- Generate a correlated 2D dataset.
- Center the data and compute principal directions.
- Project to one component, reconstruct and measure error.
- Show the SVD version as the library-friendly implementation idea.

# Notebook demo 5: PCA

**Open**

`notebooks/05_pca.ipynb`

- Generate a correlated 2D dataset.
- Center the data and compute principal directions.
- Project to one component, reconstruct and measure error.
- Show the SVD version as the library-friendly implementation idea.

# PCA: practical takeaways

| Step | Why | Check |
|---|---|---|
| Center data | remove mean offset | centered feature means near zero |
| Find components | directions of variance | explained variance ratio |
| Project | lower-dimensional representation | plot or downstream performance |
| Reconstruct | understand loss | reconstruction error |

**Mini-project link**

Compress simple 2D or 3D measurements and explain how many components are worth keeping.