Lecture 1 — Numbers in the Machine: Errors, Floating Point, and Your Workbench

Course: Numerical Methods

Book: Irfan Subakti, *Numerical Methods: From Error Analysis to Machine Learning, with Java and Python*, Chapter 1

Lecturer: Irfan Subakti — yifana@gmail.com

A computer does not calculate with the real numbers of mathematics; it calculates with a finite set of binary numbers that only approximate them. Today we set up the workbench, learn to measure error, open up the 64 bits of a double, and add up a year of laundry receipts without losing a rupiah. Book: Chapter 1, Sections 1.1–1.18.

What This Meeting Covers

1. Why numerical methods — a receipt that does not add up, and the three questions every method must answer.

2. The workbench — JDK 25, Python 3.14, VS Code, NumPy and SciPy on Windows, macOS and Linux.

3. Measuring error — true, relative and approximate error, and a stopping rule for iterations.

4. How the machine stores a number — IEEE 754 doubles, machine epsilon, and five experiments.

5. Round-off meets truncation — Taylor series, the step-size dilemma, and the laundry ledger done right.

Every program of the course exists in Java and in Python, and both versions print exactly the same output. Every output shown today is the output the programs really printed. Book: Chapter 1.

Where We Begin: a Receipt That Does Not Add Up

Bu Wati runs a small laundry near a campus. She charges Rp7,500 per kilogram; a bag of 4.1 kg costs Rp30,750.

Her nephew's spreadsheet computes 4.1 × 7500 and asks whether it equals 30750. The answer is FALSE: the product shows as 30749.999999999996.

Sixty orders a day, more than twenty thousand receipts a year. On a cheap tablet that stores amounts in a shorter format, the yearly total comes out **Rp826 short**.

Nobody typed anything wrong. The machine cannot hold 4.1 exactly, and the tiny difference survives every operation.

The same small errors live in BMKG's weather model, in a bank's instalment table and in an ojek app's distance counter. In 1991 a Patriot battery in Dhahran counted time in tenths of a second in a format that could not hold 0.1; after about 100 hours the clock was 0.34 s off and the interception failed. The error per tick was tiny. What killed was the accumulation. Book: Section 1.1.

What Numerical Methods Are

School mathematics solves $2x+3=11$ with a formula. Real problems rarely have one: the interest rate hidden in an instalment plan (Chapter 2), the depth for a given volume in a lying tank (Chapter 3).

A **numerical method** replaces the problem with a finite sequence of additions, subtractions, multiplications and divisions whose answer is as close to the truth as we need.

Every method is judged by three questions: **Does it converge? How fast? Is it stable** — do small errors stay small?

> **KEY INSIGHT.** The answer of a numerical method is never exact, but we know *how* inexact it is, and we can buy more accuracy with more work. Bisection halved 20 times knows the root to one part in a million.

These three questions are the questions of the classic textbooks, Chapra and Canale, and Burden and Faires, and they return in every chapter of the course. Book: Section 1.2.

The Workbench

Table 1.1 The tools of the workbench and the versions used throughout the book.

Tool	Role in the course
JDK 25 (LTS)	`javac` and `java`: compile and run the Java programs
Python 3.14	the interpreter, in a virtual environment `.venv`
NumPy 2.x, SciPy	library answers to check our own programs
VS Code + Java and Python extensions	editor, Run button, integrated terminal

Windows — Temurin 25 (LTS) from adoptium.net or `winget`, the Python install manager (`py install 3.14`), VS Code. **macOS** — the JDK `.pkg`, the python.org installer. **Linux** — `openjdk-25-jdk` and `python3-venv` from the distribution.

Check in a *new* terminal: `javac -version`, `python3 --version` (on Windows `py --version`), `code --version`.

Table 1.1: later patch releases work exactly the same. Ubuntu 26.04 LTS ships OpenJDK 25 and Python 3.14 by default, but installs neither `javac` nor the `venv` module unless you ask. Book: Sections 1.3–1.6.

Compiled Java, Interpreted Python

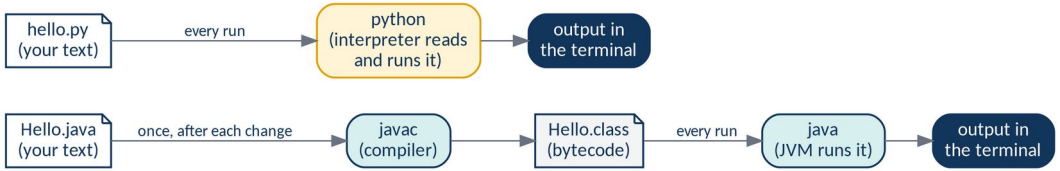


Figure 1.1 How a Java program and a Python program run. Java is compiled once by `javac` into bytecode, which the Java Virtual Machine runs; Python is read and run by the interpreter every time.

Figure 1.1: `javac Hello.java` *makes* `Hello.class`*;* `java Hello` *runs it on the JVM.* `python3 hello.py` *reads the source every time. Speed is not today's topic; correctness is. Book: Section 1.3.*

VS Code, a Virtual Environment, NumPy and SciPy

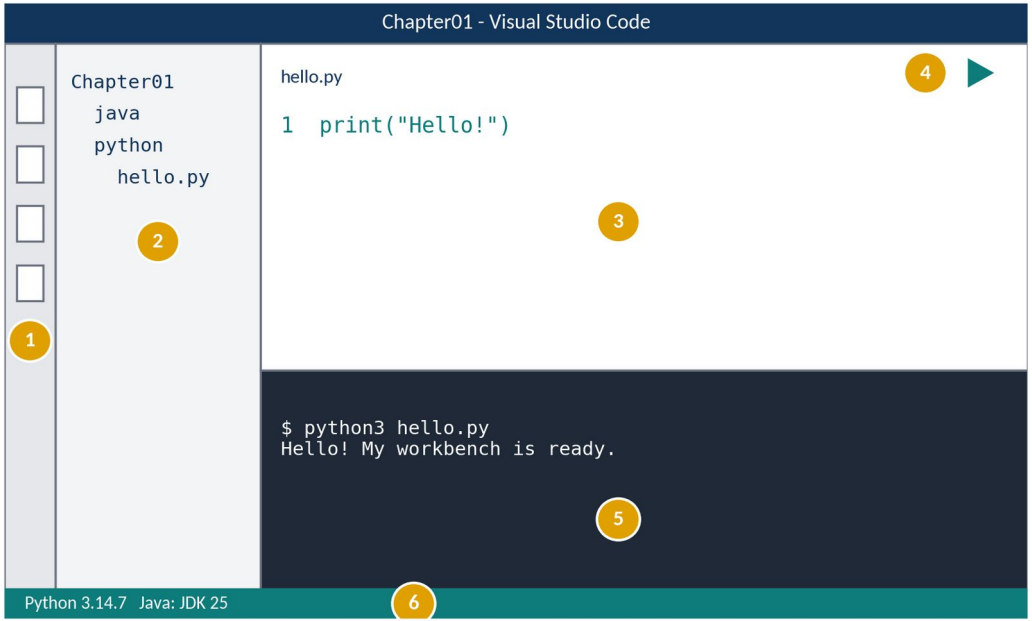


Figure 1.2 The parts of Visual Studio Code used in this book: (1) Activity Bar, (2) Explorer, (3) editor, (4) Run button, (5) integrated terminal, (6) Status Bar.

One folder per chapter; open the *folder* in VS Code. In its terminal: `python3 -m venv .venv`, activate it, then `pip install numpy scipy`.

Figure 1.2: the two Microsoft extensions, Extension Pack for Java and Python, give the Run button and the interpreter picker in the Status Bar. `check_setup.py` and `CheckSetup.java` confirm Python, the virtual environment, NumPy, SciPy and Java in one run. Book: Sections 1.7–1.8.

Measuring Error: True and Relative

If p is the true value and $\tilde{p}$ its approximation, the **true error** is

$$E_t = p - \tilde{p} \tag{1.1}$$

An error of 1 cm is huge for a screw and nothing for a road, so we scale it, giving the **relative error**:

$$\varepsilon_t = \frac{p - \tilde{p}}{p} \times 100\ \% \tag{1.2}$$

Equation (1.1) has units; Equation (1.2) has none, and a relative error near 0.5×10^{-n} means about n correct significant digits. For $\pi \approx 3.1416$ the relative error is about 2.3×10^{-6}: five to six correct digits. Book: Section 1.9.1.

Approximate Error and the Stopping Rule

In practice the truth is unknown. An iteration compares its newest answer with the previous one:

$$\varepsilon_a = \frac{\widetilde{p}_{\text{new}} - \widetilde{p}_{\text{old}}}{\widetilde{p}_{\text{new}}} \times 100\ \% \tag{1.3}$$

Stop when the absolute value of ε_a falls below the tolerance for n significant digits (Scarborough):

$$\varepsilon_s = 0.5 \times 10^{2-n}\ \% \tag{1.4}$$

Equation (1.3) and Equation (1.4) are the stopping rule of every iterative method in Chapters 2, 3, 5 and 13. For three digits the tolerance is 0.05%. Book: Section 1.9.2.

Two Kinds of Error

> **DEFINITION. Round-off error** comes from storing numbers with finitely many digits. **Truncation error** comes from cutting an infinite mathematical process short: a series, a limit, an integral.

Round-off is the machine's fault; truncation is the method's choice. Most of the course is about controlling truncation error without being ambushed by round-off.

Beside them stand blunders, model errors and data errors, which no algorithm can repair.

The step-size dilemma later today shows the two kinds of error pulling in opposite directions. Book: Section 1.9.3.

Why 0.1 Cannot Be Stored

In base 2, one tenth has a repeating expansion, just as one third does in base 10:

$$0.1 = 0.0\,0011\,0011\,0011\ldots_2 \tag{1.5}$$

A **double** (IEEE 754 binary64) keeps a sign bit, an 11-bit exponent e and a 52-bit fraction:

$$x = (-1)^s \times (1.f_1 f_2 \ldots f_{52})_2 \times 2^{e-1023} \tag{1.6}$$

Equation (1.5): the pattern 0011 never ends, so it must be cut off and rounded. Equation (1.6): the leading 1 is not stored, which gives 53 bits of precision, about 15 to 16 decimal digits. Book: Sections 1.10.1–1.10.2.

The 64 Bits of 0.1

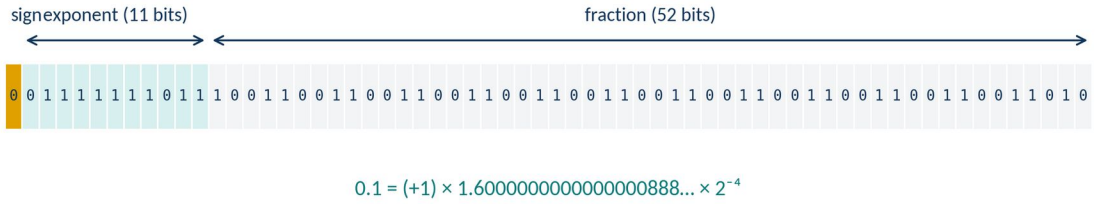


Figure 1.3 The 64 bits of the double nearest to 0.1: sign 0, exponent 01111111011 (1019, meaning 2^{-4}) and 52 fraction bits in which the pattern 1001 repeats until it is cut off and rounded.

Figure 1.3: the last bits are rounded up, so the stored value is slightly larger than 0.1. The two formats of the course are single precision (float, 32 bits, about 7 digits) and double precision (64 bits, about 16 digits). Book: Section 1.10.2.

Machine Epsilon and the Rounding Bound

Machine epsilon is the gap between 1 and the next double:

$$\varepsilon_{\text{mach}} = 2^{-52} \approx 2.22 \times 10^{-16} \tag{1.7}$$

Every stored number, and every result of $+, -, \times, \div$, is the true value times $(1+\delta)$ with a relative error δ of at most the **unit round-off** $u = 2^{-53} \approx 1.11 \times 10^{-16}$.

> **KEY INSIGHT.** One operation is almost perfect. The danger is in *many* operations, whose errors accumulate, and in *subtraction* of nearly equal numbers, which exposes errors that were already there.

Equation (1.7): in single precision epsilon is $2^{-23} \approx 1.19 \times 10^{-7}$. The model "each operation is exact, then rounded" is the basis of all error analysis in the course. Book: Section 1.10.3.

Five Experiments with FloatLab

Listing 1.1 From `float_lab.py`: adding 0.1 and 0.2, the exact stored value of 0.1, and machine epsilon found by halving.

```
    print("1. Adding 0.1 and 0.2")
    s = 0.1 + 0.2
    print(f"   0.1 + 0.2        = {s!r}")
    print(f"   0.1 + 0.2 == 0.3 : {s == 0.3}")

    print("2. What is really stored for 0.1")
    print(f"   {Decimal(0.1)}")        # the exact value of the double

    print("3. Machine epsilon, found by halving")
    eps = 1.0
    while 1.0 + eps / 2 > 1.0:     # stop when half of eps is no longer seen
        eps /= 2
```

```
1. Adding 0.1 and 0.2
   0.1 + 0.2        = 0.30000000000000004
   0.1 + 0.2 == 0.3 : False
2. What is really stored for 0.1
   0.1000000000000000055511151231257827021181583404541015625
3. Machine epsilon, found by halving
   eps              = 2.220446e-16
   1 + eps/2 == 1   : True
4. Adding 0.1 again and again
        10 times: 0.9999999999999999     error 1.110e-16
      1000 times: 99.9999999999986       error 1.407e-12
   1000000 times: 100000.00000133288     error 1.333e-06
```

Listing 1.1, and its twin `FloatLab.java`, print exactly these lines. Never compare computed doubles with `==`; use a tolerance. Experiment 4 is the Patriot clock in miniature: every addition rounds and the errors do not cancel. Book: Section 1.11.

Catastrophic Cancellation

$x^2 - 10^8 x + 1 = 0$ has a root near 10^{-8}. The school formula subtracts two numbers that agree in 16 digits:

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \tag{1.8}$$

The product of the roots is c/a (Vieta), so compute the large root safely, then

$$x_2 = \frac{c}{a x_1} \tag{1.9}$$

```
5. Subtracting nearly equal numbers: x^2 - 1e8 x + 1 = 0
   textbook formula : 7.4505805969e-09
   stable formula   : 1.0000000000e-08
   relative error of the textbook formula: 2.549e-01
```

Equation (1.8) and Equation (1.9) are equal in algebra and very different on a computer: 25% error against full accuracy. The subtraction itself is exact; it only reveals the garbage the rounding had put in the last digits. That is the meaning of numerical stability. Book: Section 1.11.

Truncation: the Taylor Series

$$f(x)=f(a)+f'(a)(x-a)+\frac{f''(a)}{2!}(x-a)^2+\cdots+\frac{f^{(n)}(a)}{n!}(x-a)^n+R_n \tag{1.10}$$

For e^x around 0 every derivative is 1, and each new term is the previous one times x/k:

$$e^x=1+x+\frac{x^2}{2!}+\frac{x^3}{3!}+\cdots \tag{1.11}$$

```
e^0.5 to 3 significant digits (tolerance 0.05 %):
terms           sum   true err %  approx err %
    1    1.000000000    39.346934
    2    1.500000000     9.020401     33.333333
    3    1.625000000     1.438768      7.692308
    4    1.645833333     0.175162      1.265823
    5    1.648437500     0.017212      0.157978
```

Equation (1.10) is the tool behind nearly every error formula of the course; the remainder R_n is the truncation error. `exp_series.py` *sums Equation (1.11) with the stopping rule Equation (1.4); the approximate error always stays above the true error, so the rule is safe. At $x=-20$ the alternating terms cancel badly, and $1/e^{20}$ is the stable route. Book: Section 1.12.1.*

The Step-Size Dilemma

The forward difference, from the Taylor series, has a truncation error proportional to h:

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{f''(\xi)}{2} h \tag{1.12}$$

h	estimate	error
1e-01	0.349241122458488	4.312e-03
1e-02	0.353112550268730	4.408e-04
1e-03	0.353509207464464	4.418e-05
1e-04	0.353548971285811	4.419e-06
1e-05	0.353552948650915	4.419e-07
1e-06	0.353553346377211	4.422e-08

Equation (1.12) for $\sqrt{x}$ at $x = 2$: each tenfold smaller step makes the error ten times smaller, as promised, but only down to about $h = 10^{-8}$. Below that the error grows again. Book: Section 1.12.2.

Where the Two Errors Balance

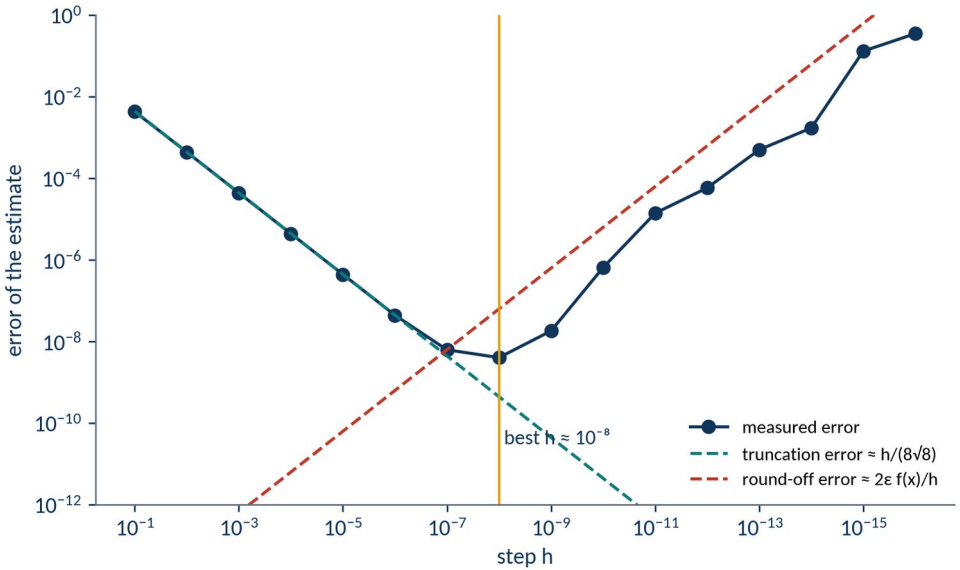


Figure 1.4 The error of the forward difference of $\sqrt{x}$ at $x=2$ against the step h: truncation error falls with h, round-off error grows as $1/h$, and the total is smallest near $h=10^{-8}$.

$$E(h) \approx \frac{M}{2} h + \frac{2\varepsilon_{\text{mach}} |f(x)|}{h} \tag{1.13}$$

$$h_{\text{opt}} = 2\sqrt{\frac{\varepsilon_{\text{mach}} |f(x)|}{M}} \tag{1.14}$$

Figure 1.4: the V shape of Equation (1.13). Minimizing it gives Equation (1.14), about the square root of machine epsilon, 10^{-8}. Smaller is not always better; Chapter 9 returns to this dilemma with better formulas. Book: Section 1.12.2.

The Laundry Ledger: Adding Up Money Safely

365 days of 60 orders, weights from 1.0 to 8.0 kg, the same pseudo-random receipts in Java and Python, summed four ways:

```
Orders in the year     : 21,900
Exact total (integer)  : Rp 739,304,250
double, simple sum     : Rp 739,304,250.00  (off by +0.00)
float, simple sum      : Rp 739,303,424.00  (off by -826.00)
float, Kahan sum       : Rp 739,304,256.00  (off by +6.00)
```

Kahan's compensated sum keeps the lost low-order part of each addition in a correction c:

$$c = \big((t + y) - t\big) - y \tag{1.15}$$

WARNING. Count money in whole units (rupiah, cents, the smallest coin unit) with integers. Use doubles for measurements, and compensated or pairwise summation for long sums.

Here is Bu Wati's Rp826. Near Rp739 million the gap between neighbouring floats is Rp64, so every one of the 21,900 additions rounds by up to Rp32. Equation (1.15) rescues most of it even in single precision. Book: Section 1.13.

Comparing with the Libraries

```
float64 eps (np.finfo)  : 2.220446e-16
float32 eps (np.finfo)  : 1.192093e-07
math.ulp(1.0)           : 2.220446e-16
next double after 1.0   : 1.0000000000000002
0.1 stored as float32   : 0.100000001490116119384765625
loop sum of 10^6 x 0.1  : 100000.00000133288
np.sum (pairwise)       : 100000.00000000003
```

NumPy 2.x confirms the machine epsilon we found by experiment, and its pairwise `np.sum` is far more accurate than a plain loop. Java gives the same facts through `Math.ulp` and `Math.nextUp`, and `BigDecimal` sums exactly. Book: Section 1.14.

Numerical Methods in Practice: Profitina

Profitina (app.profitina.com) is an AI visibility platform that measures how ready a brand's website is to be cited by AI answer engines.

Money in whole units: a Pro subscription is Rp50,000 for thirty days; the PROFIT coin has 6 decimals, so every balance is a whole number of millionths — the ledger rule at scale.

Scores that add up: combining many signals is a long sum of products; normalizing by a tiny range magnifies input error, the same effect as the $1/h$ term.

A conversion between PROFIT and rupiah is the one place where a rate meets an amount; the result is rounded once, at a clear point, by a stated rule. Book: Section 1.15.

Common Mistakes

Comparing doubles with `==`. Compare with a tolerance: relative for large numbers, absolute near zero.

Money in floating point. Use integer units; round once, by a stated rule.

Subtracting nearly equal numbers. Rearrange the formula, as in Vieta's form.

Making the step as small as possible. Below the optimal step, round-off wins.

Summing a long list in single precision. Use doubles, Kahan or pairwise summation.

In spreadsheets the same rules hold: store rupiah as whole numbers, and round only at the end with the ROUND function. Book: Sections 1.16–1.17.

Summary

1. A numerical method answers with a known, acceptable error; methods are judged by convergence, speed and stability.

2. Error is measured as true, relative and approximate error; iterations stop at ε_s (Equation (1.4)).

3. A double has 53 bits of precision; each operation errs by at most $u = 2^{-53}$, but errors accumulate and cancellation exposes them.

4. Truncation falls as the step shrinks, round-off grows; the best step balances them (Equation (1.14)).

5. Money is counted in integers; long sums use doubles and compensated summation. Our programs agree with NumPy and Java's libraries.

Everything that follows in the course, from bisection to gradient descent, is judged against these ideas. Book: Section 1.18.

Next Time

Chapter 2 — Root Finding I: Bisection and Regula Falsi. What monthly interest rate is hidden in an instalment plan?

Bracketing methods that can never fail: halve the interval, or cut it where a straight line crosses zero.

Before then: install and check the workbench, run FloatLab and the ledger in both languages, and work the exercises on relative error and single-precision sums.

The harmonic sum in single precision, forward and backward, shows once more that the order of summation matters; its solution is in the book. Book: Sections 1.19 and 1.20.

Lecture 2 — Root Finding I: Bisection and Regula Falsi

Course: Numerical Methods

Book: Irfan Subakti, *Numerical Methods: From Error Analysis to Machine Learning, with Java and Python*, Chapter 2

Lecturer: Irfan Subakti — yifana@gmail.com

Many everyday questions end in an equation that no formula can solve. Today we build the two oldest and safest ways to find a root: start from an interval in which the function changes sign and shrink it until the root is pinned down. Book: Chapter 2, Sections 2.1–2.13.

What This Meeting Covers

1. The question — the true monthly rate hidden in a phone instalment plan.

2. From question to root — the annuity formula and the loan function $f(r)$.

3. Bisection — the algorithm, its error bound, and the number of steps known in advance.

4. Regula falsi and Illinois — cutting where the chord crosses zero, and repairing the crawl.

5. Code, tests and SciPy — one module in Java and Python, unit tests, and three library answers.

The same bracketing functions are reused in later chapters. All iteration tables agree between Java and Python to the last digit. Book: Chapter 2.

Mas Dimas and the “1.67% a Month” Phone

Mas Dimas drives an online ojek in Yogyakarta. His phone is his office, and its screen has cracked for the third time.

A phone counter offers a Rp6,000,000 phone for **twelve instalments of Rp600,000**, “only 1.67% interest a month”.

The 1.67% is a **flat** rate: Rp1,200,000 interest divided by twelve months and by the full price, as if the whole debt were owed all year.

But the debt falls every month. What is the **effective** monthly rate, the one a bank would quote for the same payments?

The question “which input gives this output?” is the shape of every problem today: the rate that makes these payments fair, the depth at which a tank holds 3,000 litres, the month when a falling price reaches a given level. Book: Section 2.1.

Money Today and Money Later

At monthly rate r, an instalment A paid after k months is worth $A/(1+r)^k$ today. A fair plan makes the present values add up to the price:

$$P = \frac{A}{1+r} + \frac{A}{(1+r)^2} + \cdots + \frac{A}{(1+r)^n} \tag{2.1}$$

The geometric series sums to the **annuity formula**:

$$P = A\,\frac{1-(1+r)^{-n}}{r} \tag{2.2}$$

Equation (2.1) has first term $A/(1+r)$ and ratio $1/(1+r)$, which gives Equation (2.2). Banks use it in both directions: given r, it gives the instalment. Dimas knows P, A and n and wants r, which sits in a power and a denominator at once; no algebra isolates it. Book: Section 2.2.1.

Turning the Question into a Root

Move everything to one side and ask where it is zero:

$$f(r) = A\,\frac{1-(1+r)^{-n}}{r} - P \tag{2.3}$$

Table 2.1 The loan function Equation (2.3) at five monthly rates: it changes sign between 2% and 3%.

Rate r	Annuity factor	$f(r)$ (rupiah)
1%	11.2551	+753,046
2%	10.5753	+345,205
3%	9.9540	−27,598
4%	9.3851	−368,956
5%	8.8633	−682,049

Table 2.1: a few guesses already say a lot. The root lies between 2% and 3% — well above the advertised 1.67%. Tabulating or plotting f before running any method is the first habit of the chapter. Book: Section 2.2.2.

Why a Sign Change Guarantees a Root

DEFINITION. A **bracket** is an interval $[a, b]$ on which f is continuous and $f(a)f(b)<0$. By the Intermediate Value Theorem, f has at least one root in it.

A continuous curve that starts above the axis and ends below it must cross the axis somewhere in between.

Both conditions matter: without continuity, $1/x$ changes sign on $[-1, 1]$ and has no root at all.

KEY INSIGHT. A bracketing method turns a question with no formula into a sequence of yes-or-no questions: is the root left or right of this point? The sign of $f(c)$ is almost always computed correctly, even when its value is inaccurate. That is why these methods are so robust.

When common sense gives no bracket, walk from a starting point in steps until two neighbouring values differ in sign, or, better, plot the function first. Book: Sections 2.2.2 and 2.3.4.

The Bisection Algorithm

Pseudocode 2.1 The bisection method with a tolerance on the change of the midpoint.

```
BISECTION(f, a, b, tol, maxIter)
 1  fa = f(a);  fb = f(b)
 2  if fa * fb > 0: error "no sign change: [a, b] is not a bracket"
 3  c_old = a
 4  for k = 1 to maxIter
 5      c = a + (b - a) / 2              // the midpoint
 6      fc = f(c)
 7      if fc == 0 or |c - c_old| < tol: return c
 8      if fa * fc < 0: b = c; fb = fc   // root in [a, c]
 9      else:           a = c; fa = fc   // root in [c, b]
10      c_old = c
11  return c                             // tolerance not reached
```

Pseudocode 2.1: by hand on $[0.01, 0.05]$*, step 1 gives* $c = 0.03$ *with* $f(c) = -27{,}598$*, so the root is in* $[0.01, 0.03]$*; step 2 gives* $c = 0.02$ *with* $f(c) = +345{,}205$*, so the root is in* $[0.02, 0.03]$*. Only the sign of* $f(c)$ *decides. Book: Sections 2.3.1–2.3.2.*

Bisection on the Loan Function

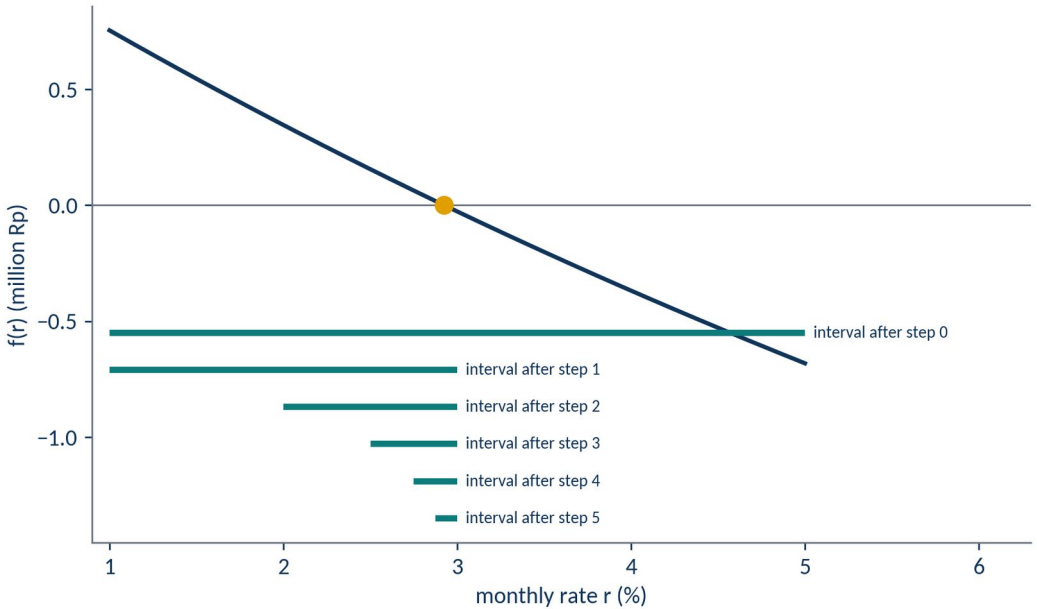


Figure 2.1 The loan function $f(r)$ of Equation (2.3) and the bracket after each bisection step, starting from $[1\ \%, 5\ \%]$: every step keeps the half that contains the sign change (amber dot).

Figure 2.1: each bar is half as wide as the one above it. The curve is smooth and falls steadily, so the root is unique in the bracket. Book: Section 2.3.

Steps Known in Advance

The k-th midpoint lies in a bracket of width $(b_0 - a_0)/2^{k-1}$, at its middle:

$$\left|c_k - r^*\right| \le \frac{b_0 - a_0}{2^k} \tag{2.4}$$

For an error below a tolerance τ it suffices that

$$k \ge \log_2 \frac{b_0 - a_0}{\tau} \tag{2.5}$$

For Dimas: $b_0 - a_0 = 0.04$, $\tau = 10^{-7}$, so $k \ge \log_2 400{,}000 = 18.6$: **19 steps**, before the program even starts.

*Equation (2.4): one binary digit per step, about 0.3 decimal digits, whatever the function looks like. Equation (2.5): each extra decimal digit costs a little more than three steps. This is **linear convergence** with rate one half. Book: Section 2.3.3.*

Regula Falsi: Cut Where the Chord Crosses

Bisection ignores the *values* of f. If $f(a)$ is large and $f(b)$ is small, the root is probably near b.
Draw the straight line through $(a, f(a))$ and $(b, f(b))$ and cut the bracket where it crosses zero:

$$c = b - f(b)\frac{a-b}{f(a)-f(b)} \tag{2.6}$$

For Dimas on $[0.01, 0.05]$ the first cut lands at $c = 0.030989$, already close to the root, where bisection's first cut was 0.03.

Equation (2.6) keeps the bracket, so regula falsi keeps bisection's guarantee that the root cannot escape. The name is Latin for "rule of false position": the line is a false function that stands in for the true one. Book: Section 2.4.1.

When One End Gets Stuck

On a curved function, one end of the bracket may never move. The error then shrinks linearly with rate

$$\lambda = 1 - \frac{f'(r^*)(a - r^*)}{f(a)} \tag{2.7}$$

The **Illinois** repair: when the same end survives twice, halve its stored function value, so the next chord leans toward it:

$$f(a) \leftarrow \frac{1}{2} f(a) \tag{2.8}$$

Equation (2.7) can be close to 1, and then regula falsi crawls. Equation (2.8) pulls the next cut across the root, the stuck end moves, and the order of convergence rises to about 1.44. In the book one algorithm serves both, with the modification switched on by a flag. Book: Sections 2.4.2–2.4.3.

One Module, Two Languages

Listing 2.1 From `roots.py`: bisection, with the trace printing left out.

```python
def bisection(f, a, b, tol=1e-7, max_iter=100, trace=False):
    """Halve [a, b] until two midpoints differ by less than tol.
    Returns (root, number of iterations)."""
    fa, fb = f(a), f(b)
    _check(fa, fb)
    c_old = a
    for k in range(1, max_iter + 1):
        c = a + (b - a) / 2                 # safer than (a + b) / 2
        fc = f(c)
        ...
        if fc == 0 or abs(c - c_old) < tol:
            return c, k
        if fa * fc < 0:                     # the root is in [a, c]
            b, fb = c, fc
        else:                               # the root is in [c, b]
            a, fa = c, fc
        c_old = c
    return c, max_iter
```

Listing 2.1: the function f is a parameter. In Python a function is an ordinary value; in Java the parameter has type `DoubleUnaryOperator`, and the Java twin returns a `Result` record with the root and the number of iterations. `_check` refuses a non-bracket. Book: Section 2.5.

Running It: Bisection

```
Price Rp 6,000,000, 12 instalments of Rp 600,000
Flat rate the seller quotes: 1.6667 % per month

Bisection on [0.01, 0.05]
  k           a           b           c           f(c)  |c - c_old|
  1  0.01000000  0.05000000  0.03000000     -27597.6039    2.000e-02
  2  0.01000000  0.03000000  0.02000000     345204.7326    1.000e-02
  3  0.02000000  0.03000000  0.02500000     154658.7589    5.000e-03
  4  0.02500000  0.03000000  0.02750000      62522.1949    2.500e-03
  5  0.02750000  0.03000000  0.02875000      17213.5840    1.250e-03
  6  0.02875000  0.03000000  0.02937500      -5253.7705    6.250e-04
```

The first rows of the run reproduce the hand computation. The table continues to exactly the 19 steps predicted by Equation (2.5), and the last column halves at every row. At step 12 the midpoint happens to be within a rupiah of the root, but bisection does not notice; it stops on the bound. Book: Section 2.6.

Running It: Regula Falsi and Illinois

```
Regula falsi on [0.01, 0.05]
  k           a           b           c           f(c)  |c - c_old|
  1  0.01000000  0.05000000  0.03098945     -62719.8812    2.099e-02
  2  0.01000000  0.03098945  0.02937568      -5278.1637    1.614e-03
  3  0.01000000  0.02937568  0.02924082       -440.7396    1.349e-04
  4  0.01000000  0.02924082  0.02922957        -36.7789    1.125e-05
  5  0.01000000  0.02922957  0.02922863         -3.0690    9.391e-07
  6  0.01000000  0.02922863  0.02922855         -0.2561    7.836e-08

Illinois on [0.01, 0.05]
  k           a           b           c           f(c)  |c - c_old|
  1  0.01000000  0.05000000  0.03098945     -62719.8812    2.099e-02
  2  0.01000000  0.03098945  0.02937568      -5278.1637    1.614e-03
  3  0.01000000  0.02937568  0.02910782       4335.4506    2.679e-04
  4  0.02910782  0.02937568  0.02922862         -2.8128    1.208e-04
  5  0.02910782  0.02922862  0.02922854         -0.0015    7.832e-08

True monthly rate : 2.9229 %
Per year, compound: 41.30 %
```

Regula falsi needs 6 steps, but look at column a: it stays at 0.01 forever — the stuck end. Illinois halves $f(a)$ at step 3, the cut jumps across the root, a finally moves, and the method finishes in 5 steps. The effective rate is 2.9229% a month, 41.30% a year, against the advertised 1.67%. Book: Section 2.6.

Three Speeds of Convergence

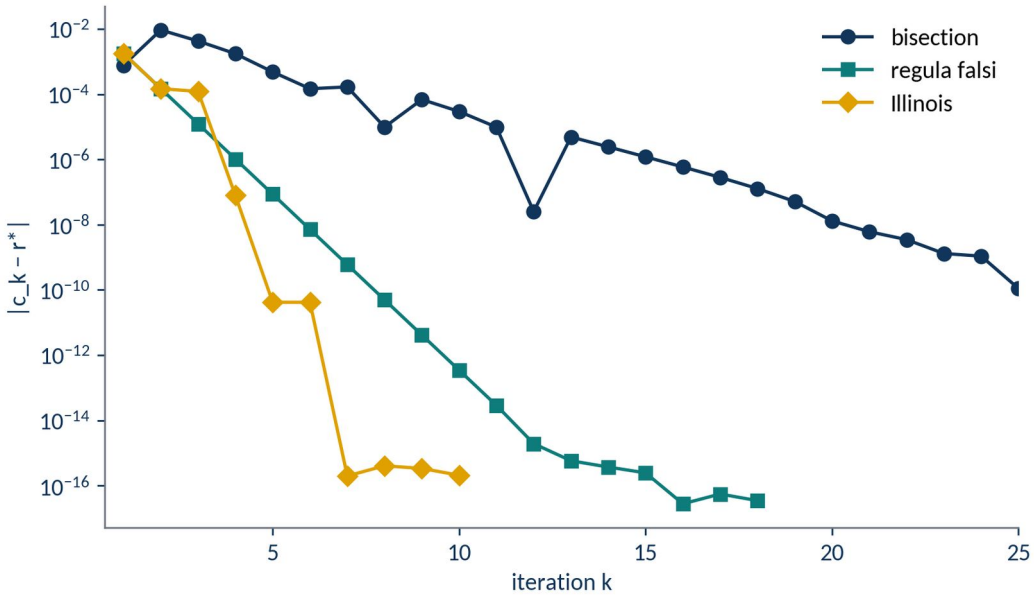


Figure 2.2 The error $\left|c_k - r^*\right|$ of the three bracketing methods on the loan function, continued past the tolerance until the error reaches the limit of double precision.

Figure 2.2: on a logarithmic scale a linearly convergent method falls along a straight line; bisection's error jumps up and down around its line of slope one half, while Illinois falls faster and faster, the mark of superlinear convergence. Book: Section 2.6.

The Debt Month by Month

Table 2.2 Dimas's debt month by month at the effective rate: the interest falls as the debt falls, and the twelfth instalment clears the debt exactly.

Month	Debt at start	Interest (2.9229%)	Principal repaid	Debt at end
1	6,000,000	175,371	424,629	5,575,371
2	5,575,371	162,960	437,040	5,138,331
6	3,749,062	109,580	490,420	3,258,642
12	582,961	17,039	582,961	0

Table 2.2 is an independent check: at the root of Equation (2.3) the twelfth instalment clears the debt exactly. The year's interest is still Rp1,200,000; only the description differs. In month 1 Dimas pays Rp175,371 interest, not Rp100,000 — the flat rate spreads interest evenly over a debt that is not even. Book: Section 2.6.

Where Regula Falsi Crawls

The worst case: $f(x) = x^{10} - 1$ on $[0, 1.3]$, root 1. The curve is flat on the left and very steep on the right.

```
bisection    : root 1.000000018 after  24 iterations
regula falsi : root 0.999999702 after  65 iterations
Illinois     : root 1.000000000 after  14 iterations
```

> **WARNING.** Plain regula falsi can stop on a small change of c while still far from the root: a crawling sequence moves little per step. Its answer here is wrong in the seventh digit although the tolerance asked for more.

`stagnation.py` and `Stagnation.java` print the same three lines. Regula falsi is slower than bisection here, and its stopping test misleads; Illinois is fastest and correct. Book: Section 2.7.

Testing It

```
ok      bisection finds sqrt 2
ok      bisection needs 21 iterations for tol 1e-6
ok      regula falsi and Illinois find sqrt 2
ok      Illinois is faster
ok      no bracket is refused
all tests passed
```

The tests use x^2-2, whose root $\sqrt{2}$ we know. For the bracket $[0,2]$ and $\tau=10^{-6}$, Equation (2.5) gives $\log_2(2\times10^6)=20.9$, so exactly 21 iterations — a test that checks the theory, not only the code. The Java tests print the same six lines. Book: Section 2.8.

Comparing with SciPy

```
scipy bisect : 2.92285919 %
scipy brentq : 2.92285408 %  (7 iterations)
toms748      : 2.92285408 %  (3 iterations)
```

`scipy.optimize` agrees with our 2.9229%. `brentq` combines bisection with interpolation and keeps the bracket's guarantee; TOMS 748 uses higher-order interpolation and needs only three iterations. The library is the professional tool; our code shows what it does inside. Book: Section 2.9.

Numerical Methods in Practice: Profitina

Profitina's **PROFIT** utility coin is designed to be deflationary: new coins are emitted on a *decaying* schedule, and coins used in the platform are burned.

When does cumulative emission reach half the ceiling? In which month does emission fall below the burn? Each asks where some function is zero.

The functions are monotone with a known bracket — exactly where bisection's guarantee Equation (2.5) is most valuable: it cannot fail, and its cost is known before it starts.

The same shape appears on the pricing side: "at what coin price does a balance cover twelve Pro renewals?" is again a monotone function and a bracket. Book: Section 2.10.

Using It for Real, and Common Mistakes

Your own plan: `python3 loan_rate.py 18000000 950000 24` for a Rp18,000,000 motorbike on 24 instalments of Rp950,000. Compare offers by the effective rate.

No bracket — tabulate first. **A pole, not a root** — check that $f(c)$ becomes small. **Undefined points** — the loan function divides by r, so keep $r = 0$ out of the bracket.

Stopping on a small $f(c)$ alone — its size depends on units. **Midpoint as $(a+b)/2$** — use $a+(b-a)/2$. **Flat rate read as effective rate** — they differ by almost a factor of two.

The same program answers any "which input gives this output?" question once f is changed: a tank depth, a break-even price, a target date. Book: Sections 2.11–2.12.

Summary

1. "Which input gives this output?" becomes $f(x)=0$; for a loan, present value minus price (Equation (2.2), Equation (2.3)).

2. A continuous f with $f(a)f(b)<0$ has a root in $[a,b]$: a bracket.

3. Bisection halves the bracket (Pseudocode 2.1); its bound (Equation (2.4)) gives the number of steps in advance.

4. Regula falsi (Equation (2.6)) is often faster but can crawl; Illinois (Equation (2.8)) repairs it, order about 1.44.

5. Dimas: 19, 6 and 5 steps, effective rate 2.9229% a month (41.30% a year) against a flat 1.67%; SciPy agrees.

Bracketing methods are slow but safe. Next time we trade some safety for much more speed. Book: Section 2.13.

Next Time

Chapter 3 — Root Finding II: Newton-Raphson, Secant and Fixed-Point Iteration. Marking the dipstick of a water truck whose tank lies on its side.

Open methods that need no bracket: they use the slope, and double the correct digits at every step — when they work.

Before then: four bisection steps by hand for $x^2 - 2$ on $[1, 2]$, and the depth of water in a spherical tank for 500 to 4,000 litres.

The spherical tank needs 21 bisection iterations for every volume, exactly as Equation (2.5) predicts; its solution in both languages is in the book. Book: Sections 2.14–2.15.

Lecture 3 — Root Finding II: Newton-Raphson, Secant and Fixed-Point Iteration

Course: Numerical Methods

Book: Irfan Subakti, *Numerical Methods: From Error Analysis to Machine Learning, with Java and Python*, Chapter 3

Lecturer: Irfan Subakti — yifana@gmail.com

Bracketing methods are safe but slow: one binary digit per step. Today we let go of the bracket and follow the shape of the function instead. The reward is speed — the correct digits double at every step — and the price is that a poor start can lead nowhere. Book: Chapter 3, Sections 3.1–3.16.

What This Meeting Covers

1. The problem — dipstick marks for a water truck whose tank lies on its side.

2. Newton-Raphson — follow the tangent; why the error is squared at each step.

3. Secant and fixed point — Newton without a derivative, and the iteration $x = g(x)$.

4. Measuring the order of convergence — 2, 1.618 and 1, seen in the numbers.

5. Failures, tests, SciPy — divergence, cycles and flat tangents; and how to choose a method.

The three open methods go into one module next to the bracketing methods of Chapter 2, in Java and in Python, with identical traces. Book: Chapter 3.

Pak Harun’s Water Truck

Every dry season wells on the slopes of **Gunungkidul** run low, and households buy water by the tank.

Pak Harun’s steel tank lies on its back: a cylinder **3.20 m long, radius 0.90 m**, a little over 8,000 litres when full.

Customers buy 500, 1,000, 2,000 or 3,000 litres. At each stop he must know how much he has let out — with a **dipstick** marked every 1,000 litres.

For a lying cylinder the marks are not evenly spaced, and the formula gives volume from depth, never depth from volume.

Equal steps on the stick would overcharge some customers and short-change others. The question, again, runs a formula backwards: which depth gives this volume? Book: Section 3.1.

The Volume of a Lying Tank

$$V(h) = L\left[R^2 \arccos\left(\frac{R-h}{R}\right) - (R-h)\sqrt{2Rh-h^2}\right] \tag{3.1}$$

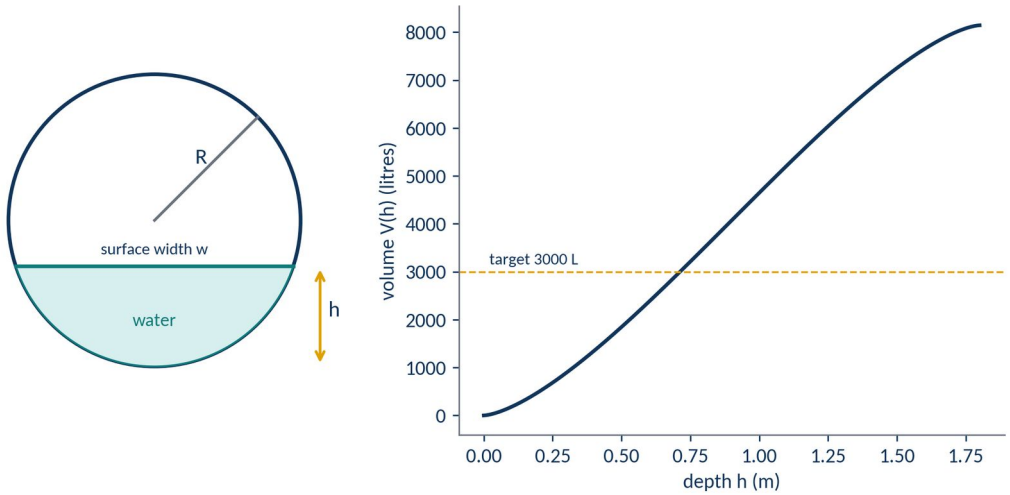


Figure 3.1 The water truck's tank seen from its end (left): the water fills a circular segment of depth h and surface width w. The volume Equation (3.1) as a function of the depth (right) is S-shaped, steepest at half depth.

Equation (3.1) is the area of a circular segment times the length. Figure 3.1: the curve is S-shaped, so the first and last thousand litres take more depth than the middle ones. Book: Section 3.2.1.

The Slope, by a Shortcut

Raise the water by a tiny Δh: the extra water is a thin slab with the area of the surface, $w \times L$. So

$$V'(h) = L\,w = 2L\sqrt{2Rh - h^2} \tag{3.2}$$

Largest at half depth, where the surface is widest; zero at the bottom and the top, where it shrinks to a line.

Equation (3.2) can be checked by differentiating Equation (3.1) term by term, but the slab argument explains why the answer is the surface width. Its zeros at the bottom and top will matter for the safeguards. Book: Section 3.2.1.

Follow the Tangent, by Hand

Box guess for 3,000 L: 3,000 of 8,143 litres is 36.8%, and 36.8% of 1.80 m is $h_0 = 0.6631$ m.

$V(0.6631) = 2.7231$ m^3: **277 litres too few**. The slope there is $V'(0.6631) = 5.5570$ m^3 per metre.

Missing 0.2769 m^3 needs about $0.2769/5.5570 = 0.0498$ m more: $h_1 = 0.7130$ m — already within **0.4 mm** of the true mark.

> **KEY INSIGHT.** Newton's idea in one sentence: at the current guess, pretend the function is its tangent line, and jump to where that line reaches the target.

The box guess is the answer a straight-line model would give; it is a sensible start, and choosing a sensible start is half of using an open method well. Book: Section 3.2.2.

Newton-Raphson

Keep the first two terms of Taylor's formula around x_k, set them to zero, solve for x:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{3.3}$$

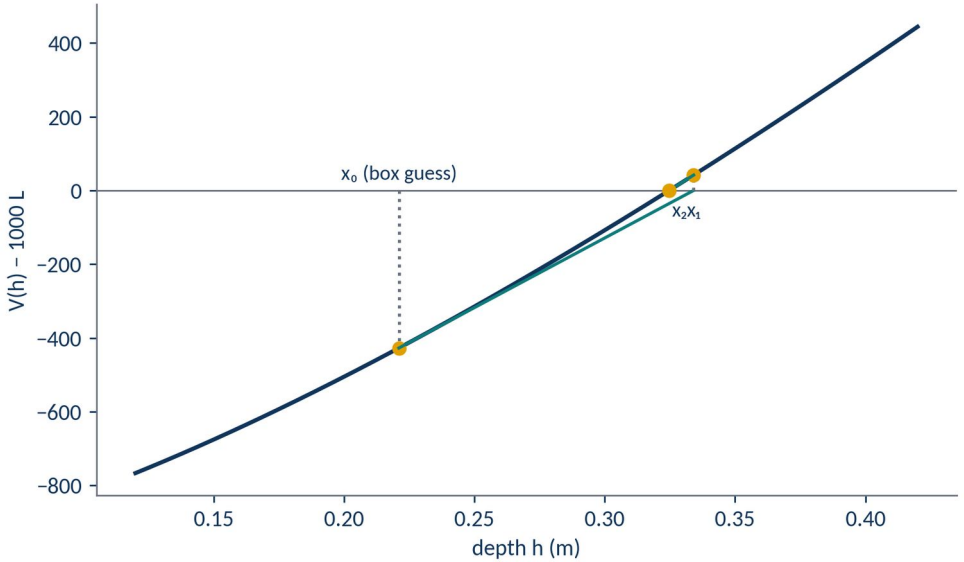


Figure 3.2 Newton's method for the 1,000-litre mark: from the box guess x_0 the tangent overshoots slightly to x_1, and the second tangent lands almost exactly on the root x_2.

Equation (3.3) drops the term with f''; that dropped term is exactly the error, as the next part of the argument shows. Figure 3.2: two tangents are nearly enough. Book: Section 3.3.1.

The Algorithm and Its Safeguards

Pseudocode 3.1 The Newton-Raphson method with its three safeguards.

```
NEWTON(f, df, x0, tol, maxIter)
 1  x = x0
 2  for k = 1 to maxIter
 3      d = df(x)
 4      if d == 0: error "f'(x) = 0: the tangent is flat"
 5      x_new = x - f(x) / d
 6      if |x_new - x| < tol: return x_new
 7      x = x_new
 8  error "no convergence in maxIter iterations"
```

Pseudocode 3.1: a flat tangent is refused, the change between iterates is the stopping test, and running out of iterations is an error. Bisection's last guess is always inside a bracket; Newton's may be anywhere. For a quadratic method the change is an excellent estimate of the error of the older iterate, so the returned one is much better still. Book: Section 3.3.1.

Why It Is So Fast

Expand f around x_k with the exact remainder at the root r and subtract Equation (3.3). With $e_k = x_k - r$:

$$e_{k+1} = \frac{f''(\xi_k)}{2f'(x_k)} e_k^2 \approx C e_k^2, C = \frac{f''(r)}{2f'(r)} \tag{3.4}$$

A method has **order** p when

$$\lim_{k \to \infty} \frac{|e_{k+1}|}{|e_k|^p} = |C| \tag{3.5}$$

Equation (3.4): an error of 10^{-3} becomes about 10^{-6}, then 10^{-12}: the number of correct digits doubles. In the language of Equation (3.5), Newton has order 2 near a simple root; bisection has order 1. Book: Section 3.3.2.

The Secant Method

No derivative? Use the slope of the line through the last two guesses:

$$x_{k+1} = x_k - f(x_k)\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} \tag{3.6}$$

Its order is the golden ratio:

$$p = \frac{1+\sqrt{5}}{2} \approx 1.618 \tag{3.7}$$

Equation (3.6) looks like regula falsi, but it keeps the latest two points, not a bracket. It is slower per step than Newton (Equation (3.7)), but each step costs only one function value, so per unit of work it is often faster. Book: Section 3.4.

Fixed-Point Iteration

Rewrite $f(x)=0$ as $x=g(x)$ and iterate:

$$x_{k+1}=g(x_k) \tag{3.8}$$

The error is multiplied by g' at each step. If the absolute value of g' is at most $K<1$ near the root, then

$$|x_k-r|\le\frac{K^k}{1-K}|x_1-x_0| \tag{3.9}$$

Equation (3.8) converges linearly with rate equal to the size of $g'(r)$ (Equation (3.9)). The same equation has good and bad forms: $x=2/x$ and $x=x^2+x-2$ are both rewritings of $x^2=2$ and both fail. Newton is the fixed-point iteration whose g' vanishes at the root. Book: Section 3.5.

Newton in Code

Listing 3.1 From `roots2.py`: the start of Newton-Raphson, with the flat-tangent safeguard.

```
def newton(f, df, x0, tol=1e-10, max_iter=50, trace=False):
    """x_new = x - f(x) / f'(x) until two iterates differ by less than tol.
    Returns (root, iterations); raises ArithmeticError if f'(x) is 0."""
    x = x0
    for k in range(1, max_iter + 1):
        d = df(x)
        if d == 0:
            raise ArithmeticError(f"f'(x) = 0 at x = {x}")
        x_new = x - f(x) / d
        if trace:
            _row(k, x_new, f(x_new), abs(x_new - x))
        if abs(x_new - x) < tol:
```

Listing 3.1: the function and its derivative are parameters; the Java twin `Roots2.java` uses two `DoubleUnaryOperator`s. `dipstick.py` passes $V(h) - \text{target}$ and $V'(h)$ from Equation (3.1) and Equation (3.2). Book: Section 3.6.

Running It: the Dipstick

```
Tank: radius 0.9 m, length 3.2 m, full 8143.0 L

Newton-Raphson for 3000 L, starting from the box guess
  k                  x           f(x)  |x - x_old|
  1     0.712968084725   2.005405e-03    4.982e-02
  2     0.712612153675   8.618449e-08    3.559e-04
  3     0.712612138378   0.000000e+00    1.530e-08
  4     0.712612138378   0.000000e+00    0.000e+00

 volume     depth  Newton  secant
 1000 L  32.47 cm       5       6
 2000 L  52.96 cm       4       5
 3000 L  71.26 cm       4       4
 4000 L  88.76 cm       3       3
 5000 L 106.21 cm       4       4
 6000 L 124.33 cm       4       5
 7000 L 144.36 cm       4       6
 8000 L 171.37 cm       5       7
```

The changes 4.98×10^{-2}, 3.56×10^{-4}, 1.53×10^{-8} *show the digits doubling. The marks are uneven: the first 1,000 litres fill 32.47 cm, the middle 1,000 only 17.45 cm. Newton needs 3 to 5 steps per mark, the secant method 3 to 7; bisection on* $[0, 1.8]$ *would need 31. Book: Section 3.7.*

Measuring the Order of Convergence

Divide two consecutive instances of Equation (3.5) and take logarithms; C drops out:

$$p \approx \frac{\ln\left(|e_{k+1}|/|e_k|\right)}{\ln\left(|e_k|/|e_{k-1}|\right)} \tag{3.10}$$

```
Newton-Raphson, x0 = 1
  k              x_k        error    ratio  order p
  1   1.50000000000000   8.579e-02
  2   1.41666666666667   2.453e-03   0.0286
  3   1.41421568627451   2.124e-06   0.0009    1.984
  4   1.41421356237469   1.595e-12   0.0000    2.000
  5   1.41421356237310   0.000e+00
```

Equation (3.10) applied to $\sqrt{2}$: Newton's measured order is 1.984, then 2.000. The same program measures about 1.6 for the secant method and 1 for fixed-point iteration, with ratio 0.1716. Theory becomes a measurement you can repeat. Book: Section 3.8.

Four Methods on One Plot

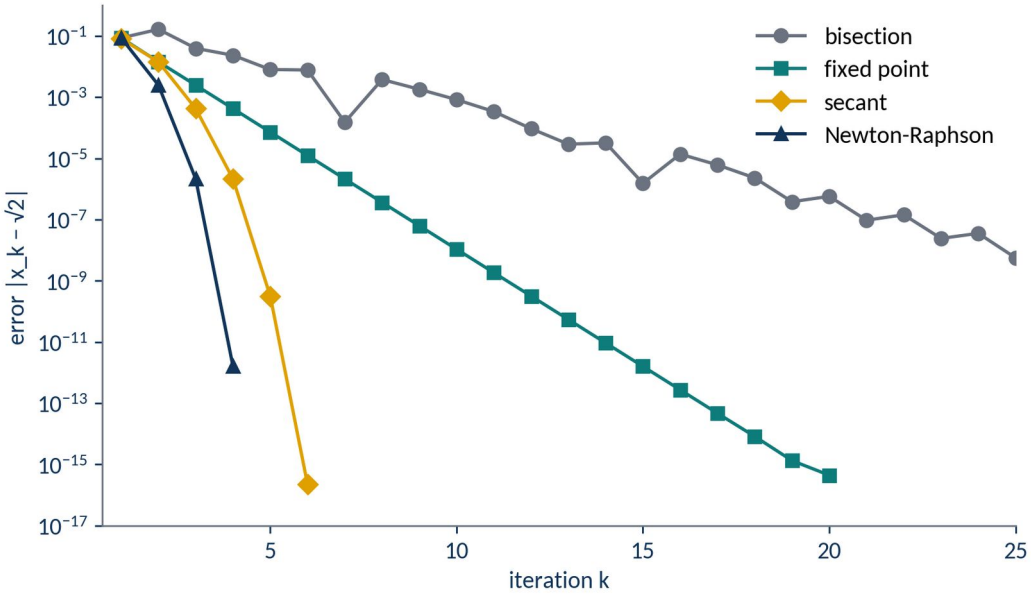


Figure 3.3 The error of four methods computing $\sqrt{2}$: bisection and fixed-point iteration fall along straight lines (linear convergence), the secant and Newton curves bend downward ever more steeply (orders 1.618 and 2). A line stops where the iterate equals the double nearest to $\sqrt{2}$.

Figure 3.3: on a logarithmic error axis a straight line means linear convergence; a curve that bends downward means superlinear. Newton reaches full precision in 5 steps, the secant method in 6, and the fixed-point iteration would need about 20. The fixed-point ratio settles at 0.1716, precisely the size of $g'(\sqrt{2})$ for the form $x = 1 + 1/(1+x)$: its error falls by the same factor at each step, forever, while Newton's error collapses. Book: Section 3.8.

When Newton Goes Wrong

```
1. f(x) = cube root of x, x0 = 0.1: each step doubles the distance
   x1 = -0.200000
   x2 = 0.400000
   x3 = -0.800000
   x4 = 1.600000
   x5 = -3.200000
2. f(x) = x^3 - 2x + 2, x0 = 0: a cycle 0, 1, 0, 1, ...
   x1 = 1.000000
   x2 = 0.000000
   x3 = 1.000000
3. f(x) = x^2 - 4x + 5 has no real root; f'(2) = 0
   stopped: f'(x) = 0 at x = 2.0
```

> **WARNING.** Newton has no guarantee. It can diverge (too steep at the root), cycle forever, stop on a flat tangent, or converge to a different root. At a double root it slows to linear convergence, halving the error per step.

`failures.py` and `Failures.java` *print the same cases; from* $x_0 = -2$ *the same cubic converges quickly. Always cap the iterations, and check that the result is in the expected range and that* f *is small there. Book: Sections 3.3.3 and 3.9.*

Testing It, and SciPy

```
ok      Newton finds sqrt 2 in at most 6 steps
ok      secant finds sqrt 2
ok      fixed point finds sqrt 2
ok      zero derivative is refused
ok      half-full tank: depth = radius
all tests passed
newton with fprime : 0.712612138378 m
newton, secant     : 0.712612138378 m
brentq on [0, 2R]  : 0.712612138378 m
```

A half-full tank must have depth exactly equal to the radius, 0.90 m — a check we can verify without any computing. SciPy's `newton` uses Newton when given `fprime` and the secant method when not; `brentq` is the bracketing method. All three agree to twelve digits. Book: Sections 3.10–3.11.

Choosing a Method

Table 3.1 The root finders of Chapters 2 and 3 compared: what each needs, how fast it converges near a simple root, and whether it is guaranteed to succeed.

Method	Needs	Order	Guaranteed?
Bisection	a bracket	1 (rate 1/2)	yes, with a known number of steps
Regula falsi	a bracket	1 (rate varies)	yes, but can crawl
Illinois	a bracket	about 1.44	yes
Secant	two starting points	about 1.618	no
Newton-Raphson	a start and f'	2	no
Fixed point	a form $x = g(x)$	1	only if g' is small

Table 3.1: in practice the best of both worlds is a hybrid — keep a bracket for safety and take open steps when they stay inside it. That is what Brent's method in `brentq` does. Book: Section 3.12.

Numerical Methods in Practice: Profitina

Many quantities in a product follow a smooth, increasing curve, and people ask questions that run the curve **backwards**.

A Pro subscription is Rp50,000 for thirty days and can be paid in PROFIT through a price oracle. Once a fee or discount depends on the amount non-linearly, “how many coins for this many rupiah?” inverts a curve — Pak Harun’s problem.

The AI Visibility Engine reports a 0–100 score; “what raw value would lift the score to 70?” inverts a normalization curve.

For smooth monotone curves with a known derivative, Newton answers in a handful of steps from a straight-line starting guess; where the curve is known only from data, the secant method needs no derivative. Book: Section 3.13.

Common Mistakes

Starting where the derivative is zero. $h=0$ gives $V'(0)=0$. Start from the box guess.

No maximum number of iterations. A cycle runs forever; cap it and raise an error.

A wrong derivative. Newton still converges, but only linearly — which hides the mistake. Check f' against a finite difference.

Fixed-point form chosen without checking g'. Compute it at the expected root first.

Expecting quadratic convergence at a multiple root. If Newton is unexpectedly slow, check whether $f'(r)$ is near zero too.

For any lying cylinder the dipstick program needs only new R and L. Measure the inside, not the outside: a steel tank with 4 mm walls has an inner radius 4 mm smaller than the outer one, and the marks move accordingly. Check the finished stick once against a known volume, for example by filling the tank from a metered tap. Book: Sections 3.14–3.15.

Summary

1. Open methods follow the function's shape; fast near a root, but they need a good start.

2. Newton follows the tangent (Equation (3.3)); its error is squared each step (Equation (3.4)).

3. Secant replaces f' by a chord slope (Equation (3.6)), order 1.618; fixed point converges linearly when g' is small.

4. Order can be measured from three consecutive errors (Equation (3.10)).

5. All eight dipstick marks in 3 to 5 Newton steps each, against 31 for bisection; SciPy agrees.

Two chapters, six root finders, one lesson: speed and safety trade against each other, and a good start buys both. Book: Section 3.16.

Next Time

Chapter 4 — Linear Systems I: Gaussian Elimination and Pivoting. How much of each fertilizer does a durian tree need?

Many unknowns at once: elimination, back substitution, and why the order of the rows matters.

Before then: two Newton steps by hand for $x^2 - 5$ from $x_0 = 2$, and Dimas's loan rate by the secant method.

The secant method finds Dimas's 2.9229% in six steps without a derivative; the solution is in the book in both languages. Book: Sections 3.17–3.18.

Lecture 4 — Linear Systems I: Gaussian Elimination and Pivoting

Course: Numerical Methods

Book: Irfan Subakti, *Numerical Methods: From Error Analysis to Machine Learning, with Java and Python*, Chapter 4

Lecturer: Irfan Subakti — yifana@gmail.com

Until now we had one unknown. Today we have many, tied together by linear equations: prices hidden in shopping receipts and the grams of four fertilizers for a durian tree. We build Gaussian elimination, count its cost, make it safe with pivoting, and learn why a small residual does not promise a small error. Book: Chapter 4, Sections 4.1–4.15.

What This Meeting Covers

1. The problem — four nutrients from four fertilizers for Bu Rahma's durian trees.

2. Elimination and back substitution — the algorithm on three receipts, and its cost of about $\frac{2}{3}n^3$.

3. Pivoting — the zero pivot, the tiny pivot, and partial pivoting.

4. Running it — the receipts and the fertilizer plan, in Java and Python.

5. Conditioning — the condition number, the Hilbert matrices, and NumPy's answer.

The module `gauss.py` / `Gauss.java` built today is reused in the chapters on iterative methods, decompositions, interpolation and least squares. Book: Chapter 4.

Bu Rahma's Durian Trees

A fruit and durian orchard near **Jember**: rambutan and mangosteen along the path, forty durian trees bearing for six years.

After a soil test the extension officer (PPL) writes what each producing tree needs per year: **480 g N, 490 g P_2O_5, 660 g K_2O, 320 g S**.

The village shop sells no pure nutrients, only bags: **NPK 15-15-15** (with 10% S), **DAP 18-46-0**, **KCl 0-0-60**, **ZA** (21% N, 24% S).

How many grams of each, so that all four nutrients come out exactly right?

Each fertilizer carries several nutrients at once, so changing one bag changes several totals. Four unknowns, four conditions: a linear system. Book: Section 4.1.

Three Receipts, Three Prices

A simpler warm-up: three receipts from the same warung show only totals (prices in thousands of rupiah):

$$\begin{aligned} 2x_1 + x_2 + 2x_3 &= 82 \\ 5x_1 + 2x_2 + x_3 &= 128 \\ 3x_1 + 2x_2 + 3x_3 &= 132 \end{aligned} \tag{4.1}$$

Separate numbers from unknowns:

$$A\,x = b\,,\; A = \begin{pmatrix} 2 & 1 & 2 \\ 5 & 2 & 1 \\ 3 & 2 & 3 \end{pmatrix},\; x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix},\; b = \begin{pmatrix} 82 \\ 128 \\ 132 \end{pmatrix} \tag{4.2}$$

Equation (4.1): rice, cooking oil and sugar. Each equation is ***linear****: every unknown is only multiplied by a number and added. Equation (4.2): row i of $A\,x = b$ is the i-th receipt; for hand work we write the augmented matrix $[\,A \mid b\,]$ and operate on its rows. Three operations never change the solution: swap two equations, multiply one by a non-zero number, add a multiple of one equation to another. Elimination uses only the first and the last. Book: Section 4.2.1.*

Elimination: Clear Each Column Below the Pivot

At stage k the **pivot** is a_{kk}. For each row i below it, the **multiplier** is

$$m_{ik} = \frac{a_{ik}}{a_{kk}} \tag{4.3}$$

and row i loses m_{ik} times row k:

$$a_{ij} \leftarrow a_{ij} - m_{ik} a_{kj} \, (j = k+1, \ldots, n), \; b_i \leftarrow b_i - m_{ik} b_k \tag{4.4}$$

Equation (4.3) and Equation (4.4): for the receipts, $m_{21} = 5/2$ and $m_{31} = 3/2$ clear column 1; then column 2 is cleared below its pivot -0.5. Subtracting a multiple of one equation from another never changes the solution. Book: Section 4.3.1.

The Staircase

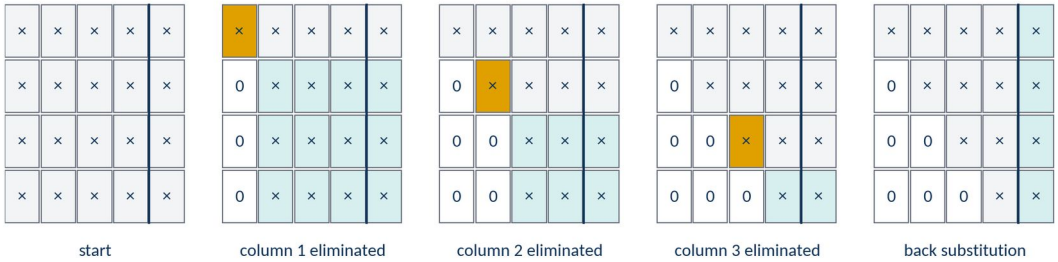


Figure 4.1 Gaussian elimination on four equations: at each stage the pivot (amber) clears its column below it and only the shaded entries change; back substitution then turns the right-hand column into the solution.

When the matrix is upper triangular, solve from the last unknown to the first — **back substitution**:

$$x_i = \frac{1}{a_{ii}}\left(b_i - \sum_{j=i+1}^{n} a_{ij}\,x_j\right), \quad i = n, n-1, \ldots, 1 \tag{4.5}$$

Figure 4.1: the work shrinks at each stage, one row and one column fewer. Equation (4.5): for the receipts, $-4\,x_3 = -68$ *gives* $x_3 = 17$*, then* $x_2 = 18$ *and* $x_1 = 15$*. Book: Section 4.3.2.*

The Algorithm

Pseudocode 4.1 Gaussian elimination with optional partial pivoting (lines 2 to 4) and back substitution.

```
GAUSS(A, b, pivot)                  // A is n x n, b has n entries
 1  for k = 1 to n - 1                          // forward elimination
 2      if pivot:
 3          p = the row i >= k with the largest |a_ik|
 4          swap rows k and p of A and b
 5      if a_kk == 0: error "zero pivot"
 6      for i = k + 1 to n
 7          m = a_ik / a_kk                     // the multiplier
 8          a_ik = 0
 9          for j = k + 1 to n: a_ij = a_ij - m * a_kj
10          b_i = b_i - m * b_k
11  if a_nn == 0: error "zero pivot"
12  for i = n downto 1                          // back substitution
13      s = b_i
14      for j = i + 1 to n: s = s - a_ij * x_j
15      x_i = s / a_ii
16  return x
```

Pseudocode 4.1: three nested loops in elimination, two in back substitution. The loops tell us the cost before we run anything. Book: Section 4.3.2.

Counting the Work

$$\text{flops}(n) = \frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{7}{6}n \approx \frac{2}{3}n^3 \tag{4.6}$$

```
  n  flops counted        formula         2n^3/3
  3            28             28             18
 10           805            805            667
 50        87,025         87,025         83,333
100       681,550        681,550        666,667
200     5,393,100      5,393,100      5,333,333
```

Equation (4.6): `op_count.py` *counts every floating-point operation and matches the formula exactly. Doubling* n *multiplies the work by eight. A laptop solves 1,000 equations in well under a second; back substitution costs only about* n^2*. Book: Section 4.3.3.*

No Unique Solution, and the Determinant

A zero pivot with only zeros below it: the matrix is **singular** — no solution, or infinitely many.

"1 kg rice and 2 L oil for Rp51,000" and "2 kg rice and 4 L oil for Rp102,000" say the same thing twice.

Elimination gives the determinant for free, with s the number of row swaps:

$$\det A = (-1)^s \prod_{k=1}^{n} a_{kk}^{(\text{final})} \tag{4.7}$$

Equation (4.7): for the receipts, pivots $2,\ -0.5,\ -4$ *and no swaps give* $\det A = 4$*. Zero means singular, but the size of a non-zero determinant says little about how trustworthy a solution is. When only* b *changes, as for trees of different ages, eliminate once and keep the multipliers — the LU idea of Chapter 6. Book: Sections 4.3.4–4.3.5.*

A Zero Pivot and a Tiny Pivot

Reorder the receipts so the first has no rice; plain elimination divides by zero at once:

$$\begin{pmatrix} 0 & 1 & 2 \\ 5 & 2 & 1 \\ 3 & 2 & 3 \end{pmatrix} x = \begin{pmatrix} 52 \\ 128 \\ 132 \end{pmatrix} \tag{4.8}$$

Worse, a *tiny* pivot gives a wrong answer without warning:

$$10^{-20} x + y = 1 \,, x + y = 2 \tag{4.9}$$

Equation (4.8) has the same prices as its solution; a person would simply use another equation first. Equation (4.9): the multiplier 10^{20} *swamps the 1 and the 2, exactly as small receipts disappear into a large float sum in Chapter 1; back substitution then returns* $x = 0$ *instead of 1. Book: Sections 4.4.1–4.4.2.*

Partial Pivoting

Before eliminating column k, bring up the row with the largest entry in that column:

$$p = \arg\max_{i \geq k} |a_{ik}|, \text{swap rows } k \text{ and } p \tag{4.10}$$

KEY INSIGHT. With partial pivoting every multiplier is at most 1 in size, so no row is ever multiplied up before it is subtracted. The swap costs no flops, and it removes both the zero pivot and the tiny pivot.

Equation (4.10) is lines 2 to 4 of Pseudocode 4.1. Wilkinson showed that entries can still grow, in the worst case by 2^{n-1}, but such matrices are so rare that elimination with partial pivoting is regarded as stable in practice; LAPACK, and therefore NumPy, uses it for general systems. Book: Section 4.4.3.

Pivoting in Code

```
1. The receipts with rice missing from the first one
   without pivoting: zero pivot in column 1
   with pivoting   : 15.000000, 18.000000, 17.000000
2. A tiny pivot: 1e-20 x + y = 1, x + y = 2 (answer near 1, 1)
   without pivoting: x = 0.000000, y = 1.000000
   with pivoting   : x = 1.000000, y = 1.000000
```

`pivot_demo.py` and `PivotDemo.java` print the same lines. The first failure announces itself; the second does not — $x = 0$ is simply wrong, and nothing in the output says so. Book: Section 4.9.

Running It: the Receipts

```
start
      2.0000     1.0000     2.0000 |    82.0000
      5.0000     2.0000     1.0000 |   128.0000
      3.0000     2.0000     3.0000 |   132.0000
after eliminating column 1
      2.0000     1.0000     2.0000 |    82.0000
      0.0000    -0.5000    -4.0000 |   -77.0000
      0.0000     0.5000     0.0000 |     9.0000
after eliminating column 2
      2.0000     1.0000     2.0000 |    82.0000
      0.0000    -0.5000    -4.0000 |   -77.0000
      0.0000     0.0000    -4.0000 |   -68.0000
rice  : Rp 15,000
oil   : Rp 18,000
sugar : Rp 17,000
largest residual: 0.000e+00
```

`receipts.py` *reads* `data/receipts.csv` *and prints every stage, which matches the hand computation.* `gauss.py` *copies the matrix and right-hand side first, so the caller's data are never changed. The residual* $b - A\,x$ *is exactly zero here. Book: Sections 4.5–4.6.*

The Durian Orchard's Fertilizer

One row per nutrient, one column per fertilizer (mass fractions), targets in grams per tree per year:

$$\begin{pmatrix} 0.15 & 0.18 & 0 & 0.21 \\ 0.15 & 0.46 & 0 & 0 \\ 0.15 & 0 & 0.60 & 0 \\ 0.10 & 0 & 0 & 0.24 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 480 \\ 490 \\ 660 \\ 320 \end{pmatrix} \tag{4.11}$$

```
fertilizer      g per tree   kg for 40
NPK 15-15-15        2171.4       86.86
DAP 18-46-0          357.1       14.29
KCl 0-0-60           557.1       22.29
ZA 21-0-0-24S        428.6       17.14
largest residual: 1.421e-14 g
```

Equation (4.11): the file has one fertilizer per line, but the matrix needs one nutrient per row — write down what rows and columns mean before building it. The program also checks each nutrient: N wanted 480.0 g, gets 480.000 g. For forty trees, Bu Rahma buys about 87 kg of NPK. Book: Section 4.7.

A Small Residual Is Not a Small Error

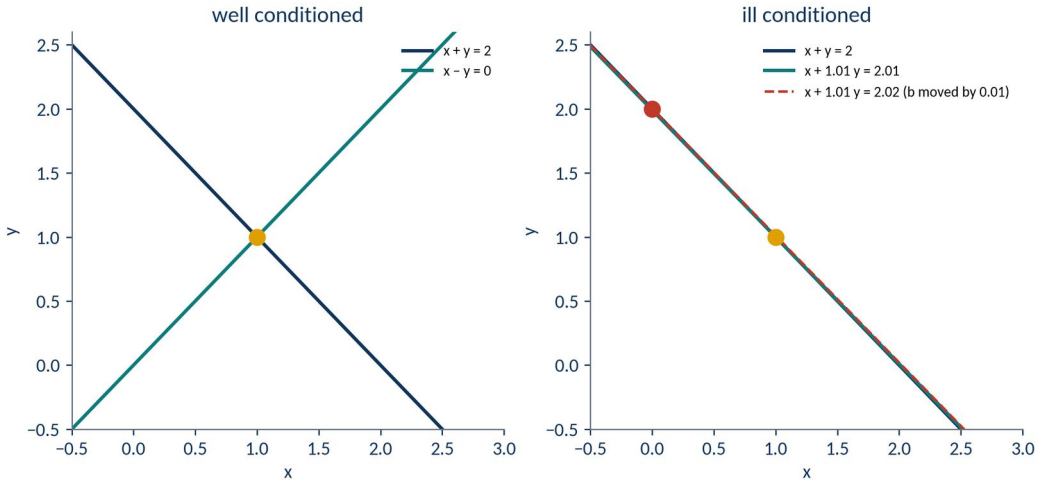


Figure 4.2 A well-conditioned system (left) and an ill-conditioned one (right): when the lines are almost parallel, a change of 0.01 in the data moves the solution from $(1, 1)$ to $(0, 2)$.

Figure 4.2: two equations are two lines, and the solution is where they cross. When the lines are almost parallel, a tiny change in the data moves the crossing point far away, while every nearby point still almost satisfies both equations. Book: Section 4.8.

The Condition Number

$$\kappa(A) = \|A\|\,\|A^{-1}\| \tag{4.12}$$

$$\frac{\|\delta x\|}{\|x\|} \le \kappa(A)\frac{\|\delta b\|}{\|b\|} \tag{4.13}$$

```
 n    cond(H)   residual  max error
 2   2.70e+01   1.11e-16   6.66e-16
 4   2.84e+04   1.67e-16   6.11e-13
 6   2.91e+07   8.33e-17   5.26e-10
 8   3.39e+10   1.67e-16   4.19e-07
10   3.54e+13   1.80e-16   3.18e-04
12   3.83e+16   4.30e-16   3.56e-01
```

Equation (4.12), Equation (4.13): relative errors in the data are magnified by up to κ; expect to lose about $\log_{10}\kappa$ digits. The Hilbert matrices show it: the residual stays at rounding level for every size, but at $n=12$ the error is 36% — nothing correct is left. Book: Section 4.8.

Testing It, and NumPy

```
ok      receipts give 15, 18, 17
ok      the input is not changed
ok      pivoting handles a zero pivot
ok      a singular matrix is refused
ok      28 flops for n = 3
all tests passed
np.linalg.solve : [2171.4  357.1  557.1  428.6]
cond(A), 2-norm : 265.7
cond(A), inf    : 446.8
Hilbert n =  6: cond 2.91e+07, max error 2.60e-10
Hilbert n = 12: cond 3.99e+16, max error 5.16e-01
```

The flop test checks Equation (4.6) at $n=3$*:* $18+13.5-3.5=28$*. NumPy's* `solve` *calls LAPACK's* `gesv`*, elimination with partial pivoting as an LU factorization, and gives the same plan. The fertilizer matrix has* $\kappa_\infty = 447$*: moderately ill-conditioned, so rounded targets change the plan considerably. Even LAPACK cannot rescue Hilbert 12. Book: Sections 4.10–4.11.*

Numerical Methods in Practice: Profitina

Profitina's AI Visibility Engine scores a website from many signals — extractable content, structured data, technical soundness — and explains each one.

A score from many signals is a weighted sum. The platform does not publish its weights, but "which weights reproduce these reference assessments?" has the shape of a linear system: one equation per reference site, one unknown per weight.

Two signals that measure nearly the same thing make the system ill-conditioned: small residual, large unstable weights, sometimes negative ones.

The remedies are the orchard's: measure the condition number, merge or drop duplicate signals, and use a stable method. With more references than weights it becomes the least-squares problem of Chapter 8. Book: Section 4.12.

Using It for Real

Your own fertilizers. Replace the lines of `fertilizer.csv` with the bags your shop sells and the targets from your soil test. A negative amount means the target cannot be met with those bags: add another (urea, “46-0-0”), or accept an approximate fit (Chapter 8).

Check the sensitivity. Change each target by its uncertainty, perhaps 5%, and solve again. Large swings mean overlapping fertilizers; single-nutrient fertilizers make the plan robust.

Mixtures everywhere. Feed for chickens or catfish from protein, energy and fibre contents, blends of rice grades to a target price: every one is $A\,x = b$.

Our programs are fine up to a few hundred unknowns. For thousands, NumPy’s `solve` uses the same method with far faster inner loops; for sparse systems with millions of unknowns, Chapter 5. Book: Section 4.13.

Common Mistakes

Elimination without pivoting. Fails on a zero pivot, silently wrong on a tiny one.

Mixing up rows and columns. Decide what row i and column j mean first.

Judging a solution by its residual. Estimate the condition number instead.

Computing the inverse to solve a system. Three times the work and less accurate; solve $A\,x = b$ directly.

Trusting the determinant's size. 10^{-3} times the identity has a tiny determinant and is perfectly conditioned.

And do not overwrite the caller's matrix silently: copy it, as our `solve` *does, or document that it is destroyed. For the orchard, round the plan to what the scale can weigh and recompute the nutrients the trees actually get. Book: Sections 4.13–4.14.*

Summary

1. n linear equations in n unknowns are $A\,x = b$ (Equation (4.2)).

2. Elimination (Equation (4.3), Equation (4.4)) makes A upper triangular; back substitution (Equation (4.5)) solves it; Pseudocode 4.1 combines both at about $\frac{2}{3}n^3$ flops.

3. Partial pivoting (Equation (4.10)) keeps every multiplier at most 1 and is stable in practice; the determinant is the signed product of pivots.

4. $\kappa(A)$ bounds the magnification of data errors; about $\log_{10}\kappa$ digits are lost; a small residual is not a small error.

5. Receipts: Rp15,000, Rp18,000, Rp17,000. Orchard: 2,171, 357, 557 and 429 g per tree, as NumPy says.

Elimination is a direct method: a fixed number of steps gives the answer. Next time we meet methods that improve a guess instead. Book: Section 4.15.

Next Time

Chapter 5 — Linear Systems II: Jacobi, Gauss-Seidel and SOR. Large, sparse systems, where elimination costs too much.

Start from a guess and improve it equation by equation; when does that converge, and how fast?

Before then: solve the rice, eggs and shallots receipts by hand, and run the Hilbert experiment yourself.

Doing one elimination by hand, with every multiplier written down, is the best way to make the algorithm your own; the solutions of all exercises are in the book. Book: Sections 4.16–4.17.

Lecture 5 — Linear Systems II: Jacobi, Gauss-Seidel and SOR

Course: Numerical Methods

Book: Irfan Subakti, *Numerical Methods: From Error Analysis to Machine Learning, with Java and Python*, Chapter 5

Lecturer: Irfan Subakti — yifana@gmail.com

Gaussian elimination costs about $\frac{2}{3}n^3$ operations and fills in the zeros of a matrix. Many real systems are large but sparse: each equation involves only a few unknowns. Today we solve them by improving a guess, sweep after sweep, and learn exactly when and how fast that works. Book: Chapter 5, Sections 5.1–5.12.

What This Meeting Covers

1. The problem — the steady temperature of a martabak griddle.

2. Three sweeps — Jacobi, Gauss-Seidel and successive over-relaxation (SOR), by hand and precisely.

3. Convergence — splittings, the spectral radius, diagonal dominance, Young's optimal ω.

4. Running it — 400 unknowns without ever building the matrix; sweep counts and a temperature map.

5. Checking and stopping — SciPy's sparse solvers, and why a small change is not a small error.

The two programs, `iterative_solvers` and `griddle`, exist in Java and Python and print exactly the same text. Book: Chapter 5.

Pak Joni’s Griddle

Pak Joni’s martabak and roti bakar cart has a square iron griddle, **40 cm on a side**.

A long burner runs under the **left** edge, a smaller one keeps the **front** edge warm; the back and right edges get no flame.

After a quarter of an hour the plate stops getting hotter: a **steady state**, heat in equals heat out.

Where is the plate between **140 °C and 170 °C**, the zone where martabak cooks best?

The edge temperatures are known from the burners and the air; the inside is unknown. The question is a map of the inside, and the map is the solution of a large linear system. Book: Section 5.1.

The Idea: Each Point Is the Mean of Its Neighbours

In a steady state the heat flowing in from the four neighbours adds up to zero, so each inner temperature is their average:

$$T = \frac{T_{\text{left}} + T_{\text{right}} + T_{\text{up}} + T_{\text{down}}}{4} \tag{5.1}$$

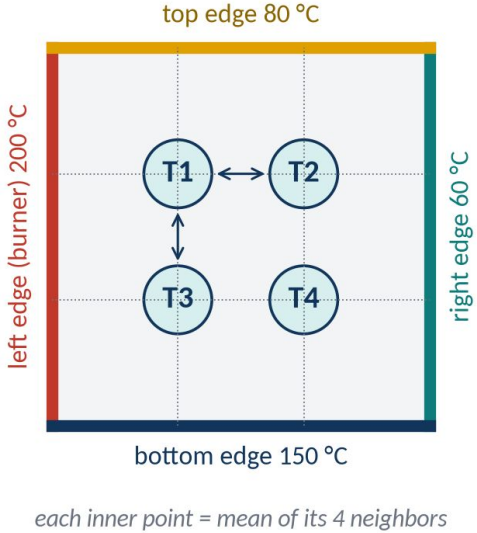


Figure 5.1 The coarsest griddle: four unknown inner temperatures, each the mean of its four neighbors; two of the neighbors are always edges with known temperatures.

Equation (5.1) is the discrete form of the heat equation at steady state; Chapter 11 derives it more carefully. Figure 5.1: with only four inner points, two neighbours of each are edges whose temperatures we know. Book: Section 5.2.1.

Four Equations

$$\begin{aligned} 4T_1 - T_2 - T_3 &= 280 \\ -T_1 + 4T_2 - T_4 &= 140 \\ -T_1 + 4T_3 - T_4 &= 350 \\ -T_2 - T_3 + 4T_4 &= 210 \end{aligned} \tag{5.2}$$

Every equation has a large diagonal coefficient 4 and at most two neighbours with coefficient -1.
On a 20×20 grid there are 400 unknowns but still at most **5 nonzeros per row**: 1,920 of 160,000 entries.

Equation (5.2) is small enough for Chapter 4, but its bigger siblings are not: elimination would fill the zeros with non-zeros and cost $\frac{2}{3}n^3$. An iterative method only ever touches the nonzeros. Book: Section 5.2.2.

Jacobi by Hand: Everybody Updates from the Old Picture

Guess 0 °C everywhere inside. Apply Equation (5.1) to every point, using the *old* guess for every neighbour:

$T_1 = (280+0+0)/4 = 70$, $T_2 = 140/4 = 35$, $T_3 = 350/4 = 87.5$, $T_4 = 210/4 = 52.5$.

Second sweep with those values: $T_1 = (280+35+87.5)/4 = 100.625$, and so on. The edges “leak” their heat inward.

Gauss-Seidel uses each new value at once, as soon as it is computed. **SOR** goes a little further than Gauss-Seidel suggests, by a factor ω.

The guess is terrible, but it does not matter. Each sweep is less wrong than the one before, because the known edge temperatures enter at every step. Book: Sections 5.2.3–5.2.5.

The Methods Precisely

Solve equation i for its own unknown. **Jacobi** uses only the previous sweep:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}\right), i = 1, \ldots, n \tag{5.3}$$

Gauss-Seidel uses the new values of the unknowns already updated:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)}\right) \tag{5.4}$$

Equation (5.3) needs two arrays, old and new; Equation (5.4) overwrites one array in place and needs less memory. Writing Jacobi into the array you read from turns it into Gauss-Seidel by accident. Book: Section 5.3.1.

Successive Over-Relaxation

$$x_i^{(k+1)} = (1-\omega)\,x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}\,x_j^{(k+1)} - \sum_{j>i} a_{ij}\,x_j^{(k)}\right) \tag{5.5}$$

Pseudocode 5.1 Successive over-relaxation; with omega = 1 it is the Gauss-Seidel method.

```
SOR(A, b, x, omega, tol, maxSweeps)
 1  for k = 1 to maxSweeps
 2      change = 0
 3      for i = 1 to n
 4          s = b[i] - (sum over j != i of A[i][j] * x[j])
 5          new = (1 - omega) * x[i] + omega * s / A[i][i]
 6          change = max(change, |new - x[i]|)
 7          x[i] = new
 8      if change < tol
 9          return x, k
10  return x, maxSweeps        // not converged: report it
```

Equation (5.5) mixes the old value with the Gauss-Seidel value; with $\omega > 1$ it overshoots on purpose, because Gauss-Seidel's corrections on this kind of problem are consistently too timid. Pseudocode 5.1 stops when no unknown moves by `tol` in a sweep, and reports failure honestly. Book: Section 5.3.1.

Matrix Splittings

Write $A = D - L - U$ (diagonal, strictly lower, strictly upper). The three methods are

$$\begin{aligned} \text{Jacobi:} \quad & D\,x^{(k+1)} = (L+U)\,x^{(k)} + b \\ \text{Gauss-Seidel:} \quad & (D-L)\,x^{(k+1)} = U\,x^{(k)} + b \\ \text{SOR:} \quad & (D-\omega L)\,x^{(k+1)} = ((1-\omega)D + \omega U)\,x^{(k)} + \omega b \end{aligned} \tag{5.6}$$

All have the form of a **stationary iteration**:

$$x^{(k+1)} = T\,x^{(k)} + c\,, T = M^{-1}N \tag{5.7}$$

Equation (5.6): each line is $M\,x^{(k+1)} = N\,x^{(k)} + c$ with $A = M - N$. A good splitting makes M easy to solve with: diagonal, or lower triangular (forward substitution). Equation (5.7) puts all three in one form, so one theory covers them. Book: Section 5.3.2.

When They Converge

Subtract the exact solution: the error is multiplied by T every sweep:

$$e^{(k+1)} = T\,e^{(k)}, \text{so } e^{(k)} = T^k e^{(0)} \tag{5.8}$$

Convergence from every start happens exactly when the **spectral radius** is below 1:

$$\rho(T) = \max_i |\lambda_i| \tag{5.9}$$

A test you can do by eye — **strict diagonal dominance**:

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}| \text{ for every } i \tag{5.10}$$

Equation (5.8) and Equation (5.9): ρ is the factor by which the error shrinks per sweep in the long run. Equation (5.10) guarantees Jacobi and Gauss-Seidel; a symmetric positive definite matrix guarantees Gauss-Seidel and SOR for $0 < \omega < 2$. Book: Sections 5.4.1–5.4.2.

How Fast: Young's Theory

For the $n \times n$ griddle the Jacobi radius is known exactly:

$$\rho_J = \cos\frac{\pi}{n+1} \tag{5.11}$$

$$\rho_{GS} = \rho_J^2, \omega_{\text{opt}} = \frac{2}{1+\sqrt{1-\rho_J^2}}, \rho\left(T_{\omega_{\text{opt}}}\right) = \omega_{\text{opt}} - 1 \tag{5.12}$$

Sweeps needed to shrink an initial error E_0 to ε:

$$k \approx \frac{\ln\left(\varepsilon / E_0\right)}{\ln \rho} \tag{5.13}$$

Equation (5.11): close to 1 for a fine grid, so Jacobi is slow. Equation (5.12): one Gauss-Seidel sweep is worth two of Jacobi, and optimal SOR is dramatically better. Equation (5.13): the sweep count grows like n^2 for Jacobi and Gauss-Seidel but only like n for optimal SOR. Book: Sections 5.4.3–5.4.4.

Running It: the Small System

```
diagonally dominant: True
Jacobi
   k       x1        x2        x3        x4     change
   1   70.0000   35.0000   87.5000   52.5000    87.5000
   2  100.6250   65.6250  118.1250   83.1250    30.6250
   3  115.9375   80.9375  133.4375   98.4375    15.3125
   4  123.5938   88.5938  141.0938  106.0938     7.6562
   5  127.4219   92.4219  144.9219  109.9219     3.8281
   6  129.3359   94.3359  146.8359  111.8359     1.9141
  converged after 27 iterations, residual 1.825e-06
  T = 131.249999 96.249999 148.749999 113.749999
```

The first two rows are the hand computation; from row 3 the change halves each time, as $\rho(T_J)=0.5$ *predicts for this system. Gauss-Seidel needs 15 sweeps, SOR with* $\omega=1.1$ *(near the optimum 1.072) needs 10, and* $\omega=1.5$ *overshoots and needs 29. The reordered system is not diagonally dominant and Jacobi diverges: change* 1.073×10^{12} *at iteration 16. Book: Section 5.6.*

Running It: 400 Unknowns

```
Griddle with 20 x 20 inner points (400 unknowns)
stop when no point moves by 1e-06 or more in a sweep
method           omega  sweeps  max error
Jacobi           1.000    1302   8.76e-05
Gauss-Seidel     1.000     683   4.31e-05
SOR              1.500     235   1.36e-05
SOR (optimal)    1.741      80   8.85e-07
```

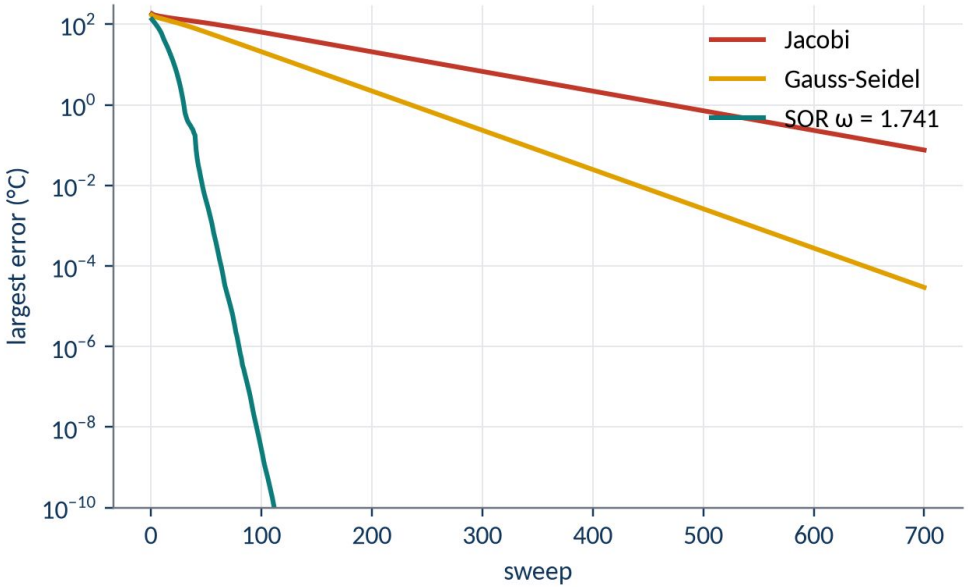


Figure 5.2 Largest error of the 20×20 griddle after each sweep, against a reference solution. Straight lines on this scale mean a constant factor per sweep: $\rho_J = 0.989$, $\rho_{GS} = 0.978$, and about 0.74 for SOR with $\omega = 1.741$.

`griddle.py` *never builds the matrix: it sweeps over a bordered grid. Figure 5.2: straight lines on a log scale, with slopes set by* ρ*. Optimal SOR needs 80 sweeps where Jacobi needs 1,302. Book: Section 5.6.*

Sweeps Against Grid Size

Table 5.1 Sweeps needed to bring the largest change below 10^{-6} on the $n \times n$ griddle. Gauss-Seidel needs about half of Jacobi's sweeps, as $\rho_{GS} = \rho_J^2$ predicts; optimal SOR grows only linearly with n.

n	Unknowns	ρ_J	Jacobi	Gauss-Seidel	SOR	ω_{opt}
10	100	0.95949	385	202	42	1.560
20	400	0.98883	1302	683	80	1.741
40	1600	0.99707	4518	2377	155	1.858

Table 5.1: doubling n roughly quadruples Jacobi's and Gauss-Seidel's sweeps but only doubles SOR's. At $n = 40$ that is 4,518 against 155. Book: Section 5.6.

Choosing Omega

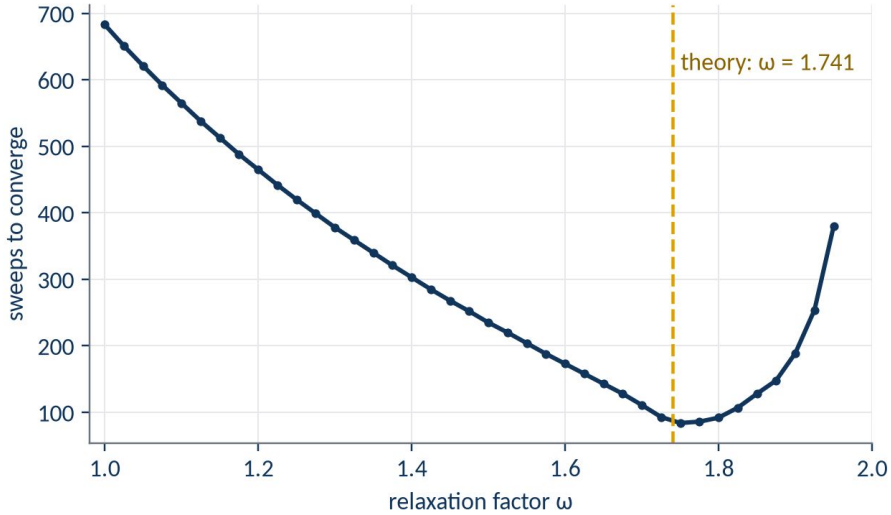


Figure 5.3 Sweeps needed by SOR on the 20×20 griddle for relaxation factors from 1 to 1.95. The minimum sits at Young's value $\omega_{\text{opt}} = 1.741$.

> **KEY INSIGHT.** The curve is lopsided: a little above the optimum is cheap (84 sweeps at 1.75, 92 at 1.80), far below it wastes most of the benefit (683 at 1.00). When in doubt, err on the high side — but stay below 2.

Figure 5.3: the scan of ω from 1 to 1.95 in steps of 0.05 finds the valley exactly where Equation (5.12) puts it. The optimum depends on n: 1.560 for $n = 10$, 1.858 for $n = 40$. Book: Section 5.6.

The Temperature Map

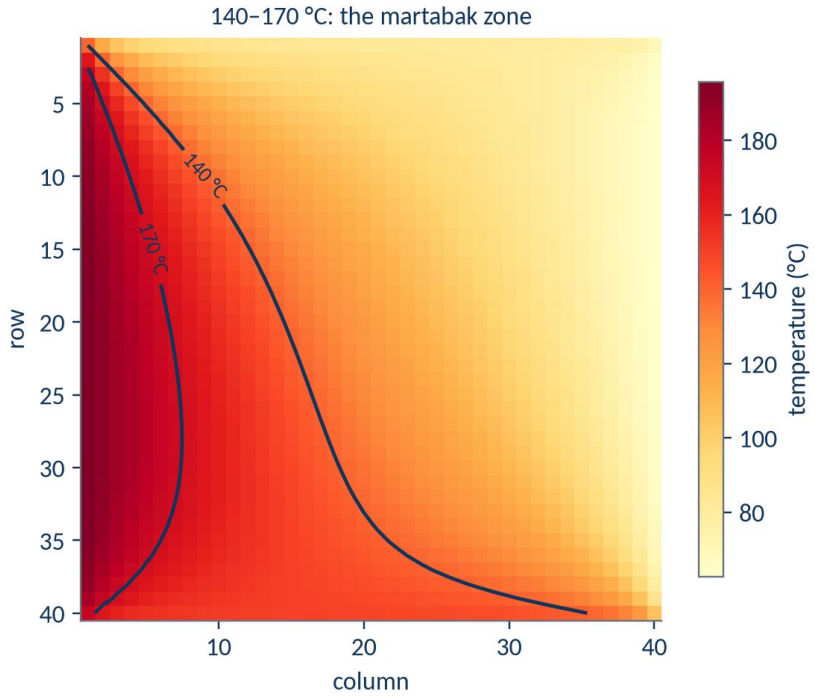


Figure 5.4 Steady temperature of the griddle on a 40×40 grid (burner on the left, warm front edge at the bottom), with the 140 °C and 170 °C lines that bound the martabak zone.

Figure 5.4: the answer to Pak Joni's question. The martabak zone is a band eight to ten centimetres wide that starts a few centimetres from the burner and widens toward the warm front edge. On the 20×20 *grid, 94 of 400 points, almost a quarter of the plate, lie in it: martabak telur in the band, roti bakar just to its right, the back right corner for keeping things warm. Book: Section 5.6.*

A Small Change Is Not a Small Error

$$\| x^{(k)} - x \| \leq \frac{\rho}{1-\rho} \| x^{(k)} - x^{(k-1)} \| \tag{5.14}$$

WARNING. For Jacobi on the 20×20 griddle, $\rho/(1-\rho)$ is about 88: when the last change is 10^{-6}, the error can still be nearly 10^{-4}. The table's error of 8.76×10^{-5} confirms it.

Equation (5.14) is the fixed-point bound of Chapter 3 in matrix form. Tighten the tolerance by the factor $(1-\rho)/\rho$, or stop on the residual together with an estimate of κ. Book: Section 5.4.5.

Checking Against the Library

```
NumPy 2.5.3, SciPy 1.18.1
4 x 4 system, numpy.linalg.solve: [131.25  96.25 148.75 113.75]
griddle matrix: 400 x 400, 1920 nonzeros of 160000
SOR (126 sweeps) vs spsolve: max difference 1.4e-12
conjugate gradient: 64 iterations, max difference 6.9e-09
```

`compare_scipy.py` builds the griddle matrix explicitly in `scipy.sparse`, solves it with the sparse direct solver `spsolve` and with the conjugate gradient method, and compares both with our SOR at a tighter tolerance. Conjugate gradient, the modern workhorse for symmetric positive definite systems, needs only 64 iterations. Book: Sections 5.7–5.8.

Numerical Methods in Practice: Profitina

Profitina's AI Visibility Engine looks at a website page by page. Pages link to one another, and a value defined by the values of neighbouring pages is a large **sparse linear system**, one unknown per page.

Warm start. Brands are re-scanned automatically at a cadence set by their tier, with a weekly score trend. From one scan to the next few pages change, so last week's values are an excellent starting guess.

Honest stopping. "The numbers have stopped moving" can hide an error ninety times larger than the last change.

Chapter 12 meets the most famous example, PageRank, a Jacobi-type iteration on the link graph of the web. A dashboard shown to a brand owner must stop on a criterion that bounds the real error. The app advises and the owner decides. Book: Section 5.10.

Common Mistakes

Iterating on a badly ordered system. Check Equation (5.10), or at least put each equation's largest coefficient on the diagonal.

Taking a small change for a small error. Use Equation (5.14).

Mixing Jacobi and Gauss-Seidel by accident. In Python, `new = x` instead of `new = x[:]` does it.

Choosing ω outside $(0, 2)$. SOR then diverges for every matrix.

Using one grid's optimal ω for another. Recompute it with Equation (5.12).

To map your own griddle, oven wall or drying floor, change four things in `griddle.py`: measure the edges with an infrared thermometer, choose the grid (one point per centimetre is plenty), handle an insulated edge (no heat crosses it), and check the model against three or four measured inner points. A computed map shows where the plate is hotter or cooler, not whether an appliance is safe. Book: Sections 5.9 and 5.11.

Summary

1. Large sparse systems arise naturally: each griddle point is the mean of its neighbours (Equation (5.1)).

2. Jacobi (Equation (5.3)), Gauss-Seidel (Equation (5.4)) and SOR (Equation (5.5)) improve a guess sweep by sweep.

3. All are stationary iterations (Equation (5.7)); they converge exactly when $\rho(T)<1$; diagonal dominance is a sufficient test.

4. On the griddle, sweeps grow like n^2 for Jacobi and Gauss-Seidel, like n for optimal SOR: 4,518, 2,377 and 155 at $n=40$.

5. Stop with Equation (5.14) in mind; SciPy's `spsolve` and conjugate gradient confirm the map.

Iterative methods trade exactness in a fixed number of steps for low memory and low cost per step. For a sparse matrix that is an excellent trade. Book: Section 5.12.

Next Time

Chapter 6 — Matrix Decompositions: LU, Cholesky and QR. Factor a matrix once, and solve with it many times.

Elimination revisited as a factorization, a faster one for symmetric positive definite matrices, and an orthogonal one that never loses accuracy.

Before then: continue the Jacobi table for sweeps 7 and 8 by hand, and measure how much the change shrinks.

The exercise asks how far T_1 is from 131.25 after sweep 8; compare your answer with the bound Equation (5.14). Solutions are in the book. Book: Sections 5.13–5.14.

Lecture 6 — Matrix Decompositions: LU, Cholesky and QR

Course: Numerical Methods

Book: Irfan Subakti, *Numerical Methods: From Error Analysis to Machine Learning, with Java and Python*, Chapter 6

Lecturer: Irfan Subakti — yifana@gmail.com

Gaussian elimination throws away its multipliers. Keep them, and elimination becomes a factorization of the matrix that can be reused for every new right-hand side. Today: LU for square systems, Cholesky for symmetric positive definite ones at half the cost, and QR for more equations than unknowns — least squares without losing digits. Book: Chapter 6, Sections 6.1–6.13.

What This Meeting Covers

1. The problem — four blocks of durian trees, one set of fertilizer bags, four recommendations.

2. LU — $PA = LU$ by hand, the Doolittle formulas, the cost of factor-once-solve-many.

3. Least squares — a fourth nutrient, the normal equations, and Cholesky.

4. QR — why the normal equations square the condition number, and Householder reflections.

5. Running and checking — two programs in Java and Python, and SciPy's factorizations.

LU, Cholesky and QR are what libraries actually do when you call `solve` or `lstsq`. Knowing them tells you which call to make and when to worry. Book: Chapter 6.

Pak Harun's Fertilizer Question

A fruit and durian orchard in **four blocks**: young trees, after harvest, before flowering, fruit filling.

Twice a year the extension officer writes grams of **N, P_2O_5 and K_2O** per tree for each block. Young trees this season: 150, 60 and 70 g.

The farm shop sells three bags: **NPK 12-12-17**, **NPK 20-10-10** and **urea 46-0-0**.

Four systems this season, four more next season — but the **matrix of grades stays the same for years**; only the right-hand side changes.

Chapter 4 could solve each system from scratch. That repeats the expensive part every time. This chapter does the expensive part once. Book: Section 6.1.

The Equations

Table 6.1 Grams of each nutrient in 100 g of each fertilizer, and the recommendation for one young tree.

Nutrient	NPK 12-12-17	NPK 20-10-10	Urea 46-0-0	Young trees need
N	12	20	46	150 g
P_2O_5	12	10	0	60 g
K_2O	17	10	0	70 g

$$\begin{pmatrix} 12 & 20 & 46 \\ 12 & 10 & 0 \\ 17 & 10 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 150 \\ 60 \\ 70 \end{pmatrix} \tag{6.1}$$

Table 6.1: measure the fertilizers in units of 100 g; each column is one bag, each row one nutrient. Equation (6.1) is the system for the young trees. Book: Section 6.2.1.

Keep the Multipliers: $PA = LU$

Pivoting brings the potash row, with 17, to the top; both other rows get the multiplier $12/17 = 0.7059$. In column 2 the nitrogen row wins (12.9412 against 2.9412), and the last multiplier is $5/22 = 0.2273$.

L = the multipliers below a unit diagonal; U = the upper triangle that elimination leaves, with diagonal 17, 12.9412, -10.4545.

$$PA = LU \tag{6.2}$$

Then every right-hand side needs only two triangular solves:

$$Ly = Pb \text{ \(forward substitution\)}, Ux = y \text{ \(back substitution\)} \tag{6.3}$$

Equation (6.2): P records the row swaps. Equation (6.3): forward substitution replays the elimination on b, back substitution is Chapter 4's. For the young trees this gives $x = (2, 3.6, 1.174)$: 200 g, 360 g and 117.4 g. Check one row by multiplying: $0.7059 \times (17, 10, 0) + (0, 12.9412, 46) = (12, 20, 46)$, the nitrogen row. Nothing new has been computed; L and U are simply elimination's work, written down instead of thrown away. Book: Sections 6.2.2–6.2.3.

Factor Once, Solve Many

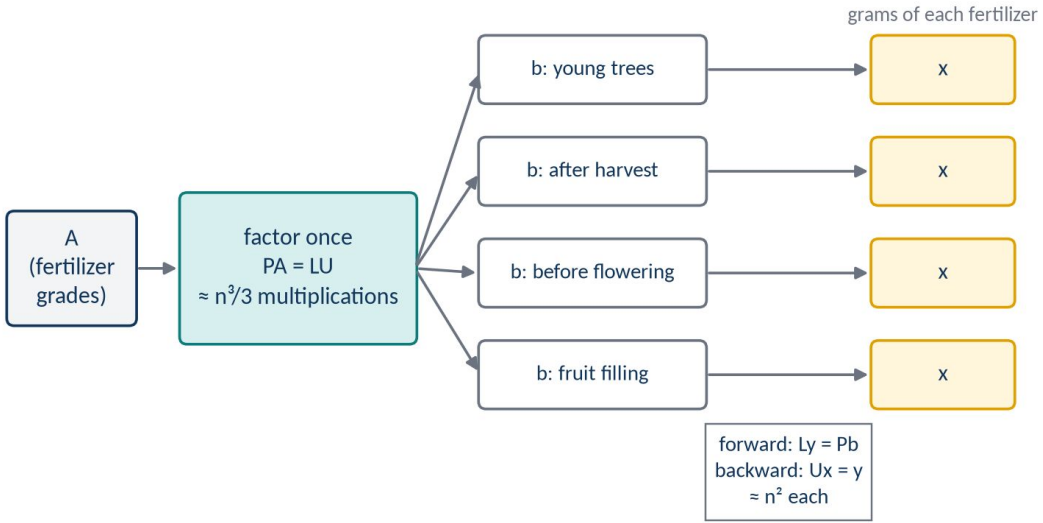


Figure 6.1 Factor once, solve many: the grade matrix is factored as $PA = LU$ once, and each block's recommendation costs only a forward and a back substitution.

Figure 6.1: the factorization depends only on the bags; the four blocks, and next season's blocks, reuse it. Book: Section 6.2.

The Doolittle Formulas and the Algorithm

$$u_{kj} = a_{kj} - \sum_{m<k} l_{km} u_{mj} \,(j \geq k), \quad l_{ik} = \frac{1}{u_{kk}} \left(a_{ik} - \sum_{m<k} l_{im} u_{mk} \right) (i > k) \tag{6.4}$$

Pseudocode 6.1 LU decomposition with partial pivoting, stored in place, and the two triangular solves.

```
LU-DECOMPOSE(A)                    // A is n x n, overwritten
 1  perm = (1, 2, ..., n); sign = 1
 2  for k = 1 to n - 1
 3      p = the row i >= k with the largest |A[i][k]|
 4      if A[p][k] == 0: error "singular"
 5      if p != k: swap rows k and p of A and of perm; sign = -sign
 6      for i = k + 1 to n
 7          A[i][k] = A[i][k] / A[k][k]           // multiplier l_ik
 8          for j = k + 1 to n
 9              A[i][j] = A[i][j] - A[i][k] * A[k][j]
10  return A, perm, sign

LU-SOLVE(A, perm, b)
11  for i = 1 to n: y[i] = b[perm[i]] - sum_{j<i} A[i][j] * y[j]
12  for i = n downto 1: x[i] = (y[i] - sum_{j>i} A[i][j] * x[j]) / A[i][i]
13  return x
```

Equation (6.4) follows from reading $A = LU$ entry by entry: row i of A is a combination of the rows of U with weights from row i of L. Pseudocode 6.1 stores L below the diagonal and U on and above it, in the same array — no extra memory. Book: Sections 6.3.1–6.3.2.

What It Costs

$$\sum_{k=1}^{n-1} (n-k)^2 \approx \frac{n^3}{3}, \text{while each solve costs} \approx n^2 \tag{6.5}$$

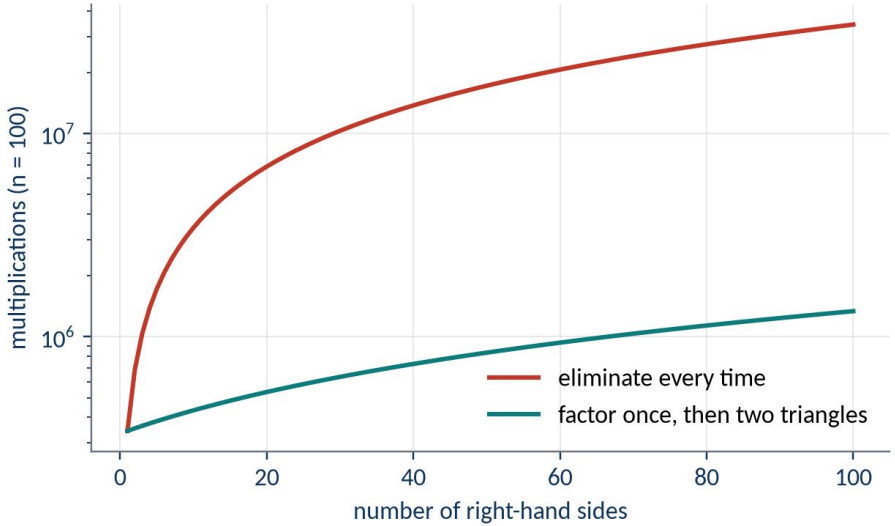


Figure 6.2 Multiplications needed for K right-hand sides with a 100×100 matrix: eliminating every time grows like $K\,n^3/3$, factoring once like $n^3/3 + K\,n^2$.

Equation (6.5) counts multiplications. Figure 6.2: for 100 right-hand sides and $n = 1000$, eliminating each time costs 33 billion multiplications, factoring once 0.43 billion. The determinant comes free as the signed product of U's diagonal, and the inverse is n solves, rarely needed. Book: Sections 6.3.3–6.3.4.

The Determinant

$$\det A = (\pm 1)\, u_{11} u_{22} \cdots u_{nn} \tag{6.6}$$

For the grades: two row swaps (17 to the top in step 1, then 12.9412 above 2.9412 in step 2), so the sign is $+1$ and $\det A = 17 \times 12.9412 \times (-10.4545) = -2300$.

> **WARNING.** Pivoting is not optional in LU either. Without it a tiny pivot produces a huge multiplier that wipes out the information in the other rows — the same failure as in Chapter 4, now stored inside L.

Equation (6.6): the sign is (-1) to the number of swaps, which Pseudocode 6.1 tracks in `sign`*. Book: Sections 6.3.4–6.3.5.*

A Fourth Nutrient: More Equations than Unknowns

Yellowing leaves: the officer adds **6 g MgO** per young tree. NPK 12-12-17 is sold as 12-12-17-2 (2% MgO); the others have none.

$$\begin{pmatrix} 12 & 20 & 46 \\ 12 & 10 & 0 \\ 17 & 10 & 0 \\ 2 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \approx \begin{pmatrix} 150 \\ 60 \\ 70 \\ 6 \end{pmatrix} \tag{6.7}$$

The first three rows force $x_1 = 2$, which delivers 4 g of MgO, not 6. No blend meets all four. Which comes **closest**?

$$\min_x \| b - A\,x \|_2^2 = \min_x \sum_{i=1}^{m} \left(b_i - \sum_{j=1}^{n} a_{ij} x_j \right)^2 \tag{6.8}$$

*Equation (6.7) is **overdetermined**. Equation (6.8) measures closeness by the sum of squared misses — the **least squares problem**, the heart of Chapter 8 as well. Book: Section 6.4.1.*

The Normal Equations

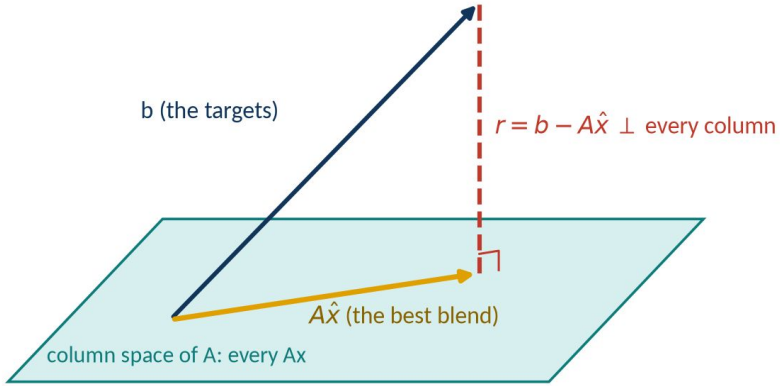


Figure 6.3 The least squares blend: $A\hat{x}$ is the point of the column space closest to the target b, and the miss $r = b - A\hat{x}$ is perpendicular to every column.

$$A^T A\hat{x} = A^T b \tag{6.9}$$

Figure 6.3: every blend Ax lies in the plane spanned by the columns; the closest point to b is the foot of the perpendicular. "Perpendicular to every column" is $A^T(b - A\hat{x}) = 0$, which is Equation (6.9): a square 3×3 system. Book: Section 6.4.2.

Cholesky: Half the Work

$S = A^T A$ is **symmetric positive definite** (SPD). It factors as $S = L L^T$:

$$l_{jj} = \sqrt{s_{jj} - \sum_{k<j} l_{jk}^2}, \quad l_{ij} = \frac{1}{l_{jj}} \left(s_{ij} - \sum_{k<j} l_{ik} l_{jk} \right) \quad (i > j) \tag{6.10}$$

For the orchard, $l_{11} = \sqrt{581} = 24.1039$, $l_{21} = 530/24.1039 = 21.9881$, and so on.

Equation (6.10) costs $n^3/6$ multiplications, half of LU, and needs no pivoting on SPD matrices because every entry is bounded by the square roots of the diagonal. A non-positive number under the square root proves the matrix is not SPD — Cholesky is also the cheapest SPD test. Book: Sections 6.4.3–6.4.4.

QR: Least Squares Without Squaring

The normal equations have a hidden cost:

$$\kappa(A^T A) = \kappa(A)^2 \tag{6.11}$$

With $A = QR$, Q orthogonal, lengths do not change, and the problem splits:

$$\|b - Ax\|^2 = \|Q^T b - Rx\|^2 = \|c_1 - R_1 x\|^2 + \|c_2\|^2 \tag{6.12}$$

Equation (6.11): if A loses 6 digits, $A^T A$ loses 12. Equation (6.12): choose x to make the first part zero, a single triangular solve $R_1 x = c_1$; the second part is the unavoidable miss. Q never has to be formed explicitly. Book: Sections 6.5.1–6.5.2.

Householder Reflections

$$H = I - \frac{2\,v\,v^T}{v^T v} \tag{6.13}$$

H reflects every vector in the mirror perpendicular to v. Choosing v well sends a whole column to a multiple of the first unit vector: all zeros below the diagonal at once.

Table 6.2 The three decompositions of this chapter, what they need, what they cost and what they are for.

Decomposition	For which matrices	Cost (multiplications)	Used for
$PA = LU$	any square	$n^3/3$	square systems, many right-hand sides
$S = LL^T$	symmetric positive definite	$n^3/6$	SPD systems, normal equations
$A = QR$	any $m \times n$, $m \geq n$	$mn^2 - n^3/3$	least squares without squaring κ

Equation (6.13): reflections are orthogonal, so they never magnify errors. Table 6.2 is the chapter on one card. Book: Sections 6.5.3–6.5.4.

Running It: LU for Four Blocks

```
max |PA - LU| = 0.0e+00
det A = -2300.0000

grams per tree           NPK 12-12-17  NPK 20-10-10   Urea 46-0-0
young trees                     200.0         360.0         117.4
after harvest                   400.0         420.0         104.3
before flowering                600.0         180.0          26.1
fruit filling                   600.0          80.0          26.1
```

`lu_blend.py` and `LuBlend.java` print P, L and U first — exactly the hand factors — then these lines. "Before flowering" and "fruit filling" differ by $(20, 10, 10)$, exactly one unit of NPK 20-10-10, so their blends differ by 100 g of that bag. Linear systems reward this kind of reading. Book: Sections 6.6–6.7.

Running It: the Least Squares Blend

```
grams per tree       normal+Cholesky  Householder QR
  NPK 12-12-17-2             224.242         224.242
  NPK 20-10-10               324.848         324.848
  Urea                       126.350         126.350
nutrient  target  delivered  miss
  N        150.0     150.00  +0.00
  P2O5      60.0      59.39  -0.61
  K2O       70.0      70.61  +0.61
  MgO        6.0       4.48  -1.52
residual norm 1.7408
```

Both methods agree here. Nitrogen is met exactly, because urea contains nothing but nitrogen and can fix that row alone; the unavoidable miss is spread over phosphate, potash and magnesium where it costs least in squared grams. Whether 1.5 g of MgO matters more than 0.6 g of phosphate is an agronomic question; weighting the rows tells the program. Book: Section 6.7.

When the Normal Equations Fail

```
Polynomial columns t^0..t^d at 20 points, true x = all ones
 d   normal+Cholesky error   Householder QR error
 3             8.14e-13              7.77e-16
 6             4.14e-09              8.35e-13
 9             1.45e-03              4.05e-10
12    fails (pivot <= 0)              2.81e-08
```

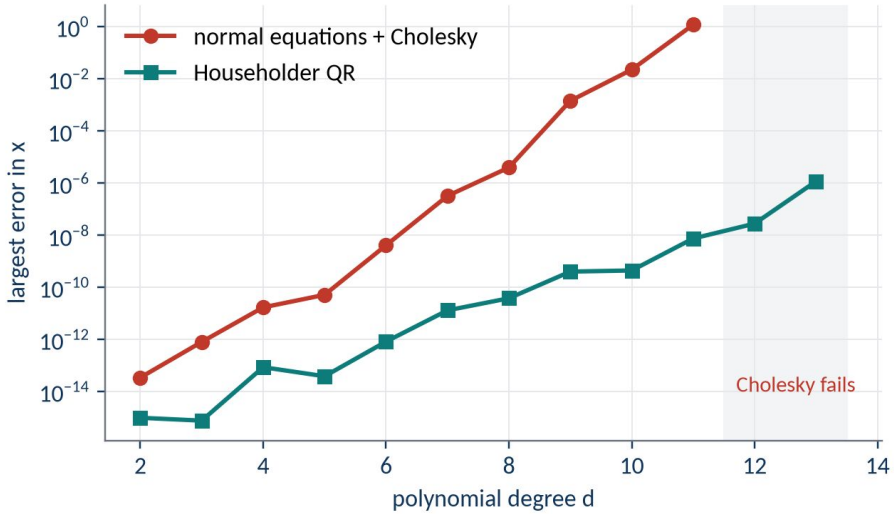


Figure 6.4 Largest error of the least squares solution for polynomial columns of degree d at 20 points: the normal equations lose about twice as many digits as Householder QR, and from degree 12 Cholesky fails.

Figure 6.4: Equation (6.11) made visible. At degree 9 QR keeps ten digits where the normal equations keep three; at degree 12 rounding makes $A^T A$ look not positive definite and Cholesky stops, while QR still gives eight digits. Book: Section 6.7.

Checking Against the Library

```
NumPy 2.5.3, SciPy 1.18.1
scipy.linalg.lu: U diagonal [ 17.      12.9412 -10.4545]
lu_solve vs ours, all four blocks: max difference 6.2e-15
numpy.linalg.lstsq grams: [224.242 324.848 126.35 ]  residual norm 1.7408
cho_solve grams:          [224.242 324.848 126.35 ]
numpy.linalg.qr R diagonal: [-24.1039  10.7946 -10.1559]
degree 9 columns: cond(A) = 3.79e+06, cond(A^T A) = 1.44e+13
```

SciPy reproduces every number. Note the convention: `scipy.linalg.lu` *returns* $A = PLU$*, the transpose of our* P*. NumPy's* R *has the same diagonal as our Cholesky factor up to sign —* R *and* L^T *are the same matrix in exact arithmetic. The last line is Equation (6.11) measured:* 3.79×10^6 *squared is* 1.44×10^{13}*. Book: Section 6.8.*

Numerical Methods in Practice: Profitina

Profitina's AI Visibility Engine turns measured signals into a readiness score and explains each signal. Choosing weights so the score agrees with what is observed is, in shape, an **overdetermined least squares problem**: one row per observed site, one column per signal.

Signals that measure nearly the same thing make $\kappa(A)$ large and $\kappa(A^T A)$ enormous; the normal equations can then give weights that are pure rounding noise. **QR is the safe road**, and libraries use it by default.

When one model is evaluated for many brands at every scheduled re-scan, the expensive factorization belongs outside the loop and only the cheap substitutions inside it, as in Figure 6.1. Book: Section 6.11.

Making It Better, and Common Mistakes

Negative grams. Nothing in LU or QR knows fertilizer cannot be negative; a recommendation of (150, 100, 120) gives -4.35 g of urea. Use nonnegative least squares, `scipy.optimize.nnls`.

Computing the inverse to solve a system. Three times the cost of `solve`, and less accurate.

Refactoring for every right-hand side. Factor once, outside the loop.

Normal equations on nearly dependent columns. They square κ; use QR.

To use `lu_blend.py` for your own orchard, rice field or vegetable garden, change the grades of your bags, the recommendations, and the names. Book: Sections 6.9, 6.10 and 6.12.

Summary

1. Elimination with its multipliers recorded is $PA = LU$ (Equation (6.2), Pseudocode 6.1), from the Doolittle formulas (Equation (6.4)).

2. Factoring costs $n^3/3$; each right-hand side then costs about n^2 (Equation (6.3), Equation (6.5)).

3. More requirements than unknowns lead to least squares (Equation (6.8)); the best residual is perpendicular to the columns, giving the normal equations (Equation (6.9)).

4. Cholesky (Equation (6.10)) solves SPD systems at half the cost of LU.

5. The normal equations square κ (Equation (6.11)); Householder QR (Equation (6.13), Equation (6.12)) does not.

The Java and Python programs agree byte for byte, and SciPy and NumPy reproduce every number. Book: Section 6.13.

Next Time

Chapter 7 — Interpolation: Lagrange, Newton and Splines. A curve through measured points, and values between them.

One polynomial through all points, its danger at the ends, and the smooth piecewise alternative.

Before then: solve for the “before flowering” block by hand with the factors P, L and U, $b = (120, 90, 120)$.

Check your hand answer against the program’s 600, 180 and 26.1 g. The solutions of all exercises, including the inverse of the grade matrix, are in the book. Book: Sections 6.14–6.15.

Lecture 7 — Interpolation: Lagrange, Newton and Splines

Course: Numerical Methods

Book: Irfan Subakti, *Numerical Methods: From Error Analysis to Machine Learning, with Java and Python*, Chapter 7

Lecturer: Irfan Subakti — yifana@gmail.com

A table gives values at a few points; life asks for the values in between. Today we build functions through tabulated points — a straight line, one polynomial through all of them, and a smooth chain of cubics — and measure honestly which one reads a fuel tank best. Book: Chapter 7, Sections 7.1–7.12.

What This Meeting Covers

1. The problem — a fuel kiosk’s dipstick and a wall chart with a mark every 20 cm.

2. Polynomial interpolation — the Lagrange and Newton forms, divided differences, Horner evaluation.

3. The error and its trap — the error formula, Runge’s function, Chebyshev points, Hermite.

4. Cubic splines — curvatures from a tridiagonal system, the Thomas algorithm, end conditions.

5. Measuring it — every method against the exact tank formula, and against NumPy and SciPy.

The tank is a perfect teacher because, unlike most tables, its exact function is known: we can measure every error instead of guessing it. Book: Chapter 7.

Pak Darto’s Dipstick

A small **fuel kiosk**: a steel tank lying on its side, 200 cm long and 120 cm across, read with a wooden dipstick.

Every evening the depth is converted to litres and compared with what the pump says it sold. A disagreement means a leak, a faulty pump, or a sale nobody wrote down.

The wall chart shows every **20 cm**; the stick says 47 cm, or 13 cm. For years Pak Darto has drawn a straight line between the nearest marks in his head.

Near the bottom and top, that straight-line reading is wrong by **more than 30 litres**.

A lying tank is not a bucket: the first 10 cm hold very little because the bottom is narrow, while 10 cm in the middle hold a lot. His nephew suspects that some of the “missing” litres come from the reading, not from the tank. Book: Section 7.1.

The Exact Tank, Our Measuring Rod

$$V(h) = L\left[r^2 \arccos\left(\frac{r-h}{r}\right) - (r-h)\sqrt{2rh-h^2}\right] \tag{7.1}$$

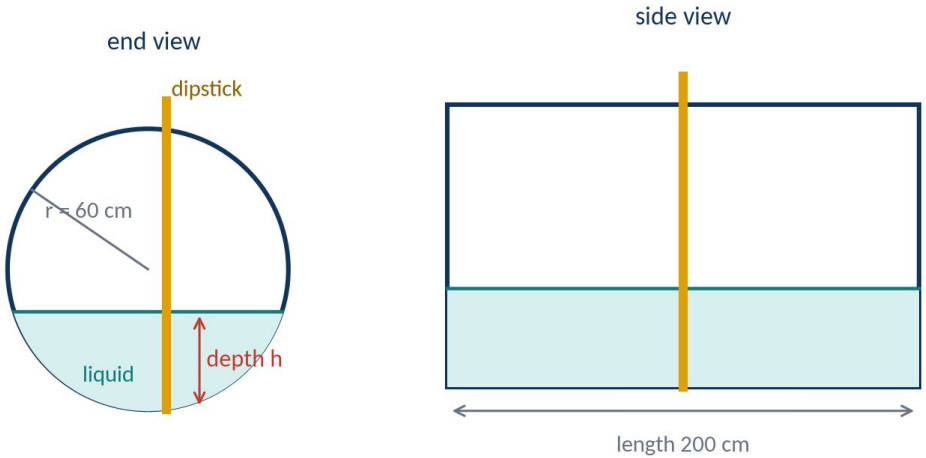


Figure 7.1 The lying tank of the fuel kiosk: the dipstick measures the depth h, and the volume is the area of the circular segment times the length.

Equation (7.1) is the same formula as Chapter 3's water truck. Here we pretend we know only the table, build interpolants from it, and use Equation (7.1) to measure how wrong each one is. Figure 7.1: the segment of the circle below the water line. Book: Section 7.1.

The Idea: Straight Lines Between Marks

$$p(x) = y_i + \frac{x - x_i}{x_{i+1} - x_i}(y_{i+1} - y_i) \tag{7.2}$$

Its error is at most $\frac{h^2}{8}$ times the largest curvature of f on the piece.

In the middle of the tank the curve is almost straight: error about 3.4 L. Near the ends it bends sharply: about **34 L.**

Equation (7.2) is what Pak Darto does in his head. The error bound says where to worry: where the function bends. Two ways out: use more information per piece (a curve instead of a line), or more marks. Book: Section 7.2.

One Polynomial Through All Points: Lagrange

Through $n+1$ points with distinct x_i there is exactly one polynomial of degree at most n. Lagrange writes it with basis polynomials that are 1 at one node and 0 at the others:

$$\ell_i(x)=\prod_{j\neq i}\frac{x-x_j}{x_i-x_j},\quad p(x)=\sum_{i=0}^{n} y_i\,\ell_i(x) \tag{7.3}$$

Equation (7.3) is easy to write down and to prove correct: at node x_k every term vanishes except $y_k\cdot 1$. It is awkward to extend — a new point changes every basis polynomial. Book: Section 7.3.1.

The Newton Form and Divided Differences

$$p(x)=f[x_0]+f[x_0,x_1](x-x_0)+f[x_0,x_1,x_2](x-x_0)(x-x_1)+\cdots+f[x_0,\ldots,x_n]\prod_{j=0}^{n-1}(x-x_j) \tag{7.4}$$

$$f[x_i,\ldots,x_{i+k}]=\frac{f[x_{i+1},\ldots,x_{i+k}]-f[x_i,\ldots,x_{i+k-1}]}{x_{i+k}-x_i} \tag{7.5}$$

Equation (7.4) is the same polynomial written so that one more point adds one more term. Its coefficients are the ***divided differences*** *Equation (7.5), computed column by column like a table of slopes of slopes. Book: Section 7.3.2.*

The Divided-Difference Table by Hand

Table 7.1 The divided-difference table for the first four marks; the top entry of each column is a coefficient of the Newton form.

x_i	$f[x_i]$	First	Second	Third
0	0.00	12.39	0.205525	−0.00220188
20	247.80	20.611	0.0734125	
40	660.02	23.5475		
60	1130.97			

Evaluate by nesting — **Horner's scheme**, n multiplications:

$$p(x) = c_0 + (x - x_0)\big(c_1 + (x - x_1)\big(c_2 + \cdots + (x - x_{n-1})\,c_n\big)\big) \tag{7.6}$$

Table 7.1: for example $247.80/20 = 12.39$ and $(20.611 - 12.39)/40 = 0.205525$. The cubic is $p_3(x) = 12.39\,x + 0.205525\,x(x-20) - 0.00220188\,x(x-20)(x-40)$. Equation (7.6) evaluates it from the inside out. Book: Sections 7.3.2–7.3.3.

The Error of Polynomial Interpolation

$$f(x) - p(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=0}^{n} (x - x_i) \tag{7.7}$$

Two factors: a **derivative** factor that belongs to the function, and a **node polynomial** that belongs to our choice of points.

The node polynomial is small in the middle of equally spaced nodes and **large near the ends**.

Equation (7.7) explains both successes and disasters. Outside the table the node polynomial grows rapidly: that is why extrapolation is dangerous. Book: Section 7.3.4.

Runge's Warning

$$f(x)=\frac{1}{1+25x^2},\ -1\le x\le 1 \tag{7.8}$$

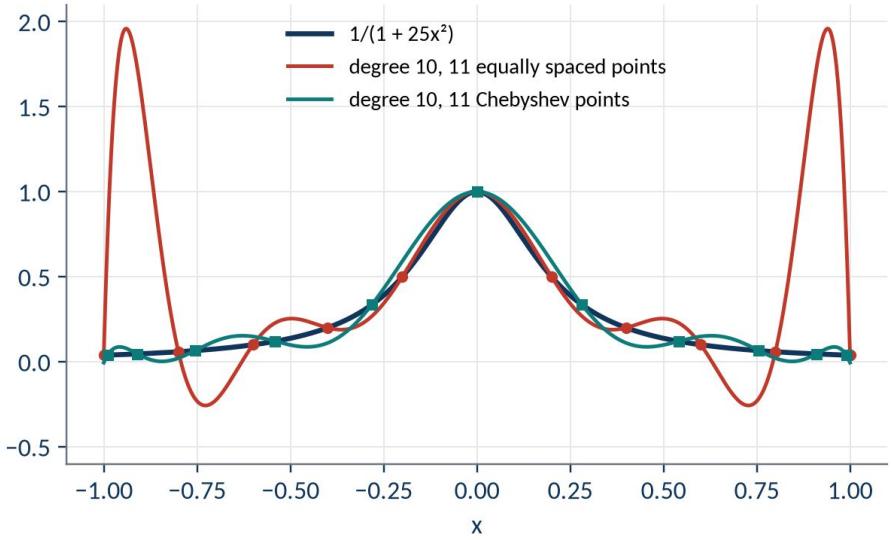


Figure 7.2 Runge's function and its degree-10 interpolants: through 11 equally spaced points the polynomial swings wildly near the ends; through 11 Chebyshev points it follows the function closely.

$$x_i = \cos\frac{(2i+1)\pi}{2n+2}, i = 0, 1, \ldots, n \tag{7.9}$$

Equation (7.8) is smooth and harmless-looking, yet Figure 7.2 shows equally spaced interpolation getting worse as the degree grows. ***Chebyshev points*** *Equation (7.9) crowd toward the ends, where the node polynomial is large, and keep the error uniformly small. Book: Section 7.3.5.*

Hermite: When Slopes Are Known Too

Match values *and* slopes s_0, s_1 at two nodes h apart: a cubic. At the midpoint

$$p\left(\frac{x_0+x_1}{2}\right)=\frac{y_0+y_1}{2}+\frac{h}{8}\left(s_0-s_1\right) \tag{7.10}$$

Equation (7.10): the straight-line midpoint plus a correction from the difference of the slopes. For the tank the slope is the surface width times the length, which we know from Chapter 3; at the bottom of the tank Hermite cuts the straight-line error by two thirds. Book: Section 7.3.6.

Running It: Line Against Polynomial

```
depth   table   linear   error  Lagrange   Newton   error
   10    90.03   123.90   33.87     93.62    93.62    3.59
   30   442.21   453.91   11.70    441.62   441.62   -0.59
   50   892.09   895.50    3.40    892.37   892.37    0.28
   70  1369.86  1366.45   -3.41   1369.57  1369.57   -0.29
   90  1819.73  1808.04  -11.69   1820.34  1820.34    0.61
  110  2171.92  2138.05  -33.87   2168.32  2168.32   -3.60
largest error on 0..120 cm: linear 34.49 L, degree 6 5.45 L
```

`interp.py` and `Interp.java` test at the six 10 cm lines the wall chart does not have. Lagrange and Newton print the same numbers, as they must: two ways of writing one polynomial. The degree-6 polynomial is 10 times better than the straight line near the ends. Its last Newton coefficient, 2.17×10^{-12}*, would be zero for exact data; it is the table's rounding to two decimals, amplified. Book: Sections 7.5–7.6.*

The Error Curves

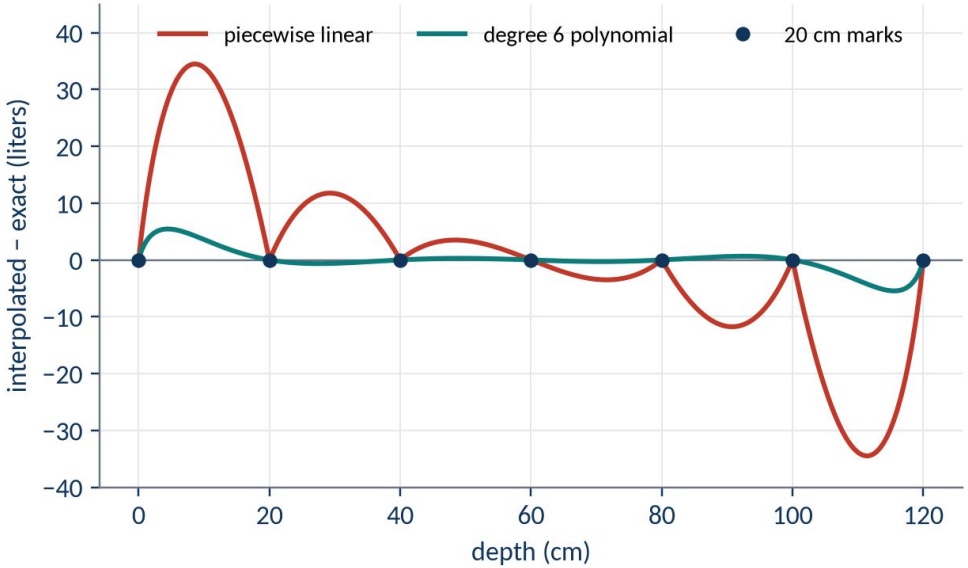


Figure 7.3 Error of piecewise linear and degree-6 polynomial interpolation from the 20 cm marks, measured against the exact tank formula.

```
Runge's function 1/(1+25x^2) on [-1, 1]: largest error
   n   equally spaced   Chebyshev points
   5           0.4327             0.5559
  10           1.9156             0.1092
  15           2.1076             0.0831
  20          59.8223             0.0153
```

Figure 7.3: each straight piece has its own arch, tallest near the ends; the polynomial is almost perfect in the middle and wobbles at the ends. Runge's experiment in numbers: at $n = 20$ equally spaced points give an error of 59.8, Chebyshev points 0.015. Book: Section 7.6.

Cubic Splines: a Smooth Chain of Cubics

One cubic per interval, joined so that value, slope and **curvature** M_i match at every node. With $a = x_{i+1} - x$, $b = x - x_i$:

$$S(x) = \frac{M_i a^3 + M_{i+1} b^3}{6 h_i} + \left(\frac{y_i}{h_i} - \frac{M_i h_i}{6} \right) a + \left(\frac{y_{i+1}}{h_i} - \frac{M_{i+1} h_i}{6} \right) b \tag{7.11}$$

Matching slopes gives one equation per inner node, with Δ_i the slope of piece i:

$$h_{i-1} M_{i-1} + 2(h_{i-1} + h_i) M_i + h_i M_{i+1} = 6(\Delta_i - \Delta_{i-1}), \; i = 1, \ldots, n-1 \tag{7.12}$$

Equation (7.11) matches the values and the curvatures by construction. Equation (7.12) is tridiagonal and strictly diagonally dominant — Chapter 5's friendliest case. Two conditions are missing: ***natural*** *(*$M = 0$ *at the ends),* ***clamped*** *(known end slopes) or* ***not-a-knot****. Book: Sections 7.4.1–7.4.2.*

The Thomas Algorithm

$$d_i \leftarrow d_i - \frac{l_i}{d_{i-1}} u_{i-1}, r_i \leftarrow r_i - \frac{l_i}{d_{i-1}} r_{i-1} \left(i = 1, \ldots, n\right); M_n = \frac{r_n}{d_n}, M_i = \frac{r_i - u_i M_{i+1}}{d_i} \tag{7.13}$$

Gaussian elimination that knows the matrix has only three diagonals: **linear time** and no pivoting needed.

By hand, natural spline at 10 cm: $M_0 = 0$, $M_1 = 0.5990$ gives $S\left(10\right) = 4.99 + \left(12.39 - 2.00\right) \times 10 = 108.93$ L.

Equation (7.13): a forward sweep and a backward sweep, about $8n$ operations. Diagonal dominance guarantees that no d_i becomes zero. Book: Section 7.4.3.

Running It: the Splines

```
curvatures M_i = S''(x_i) at the 20 cm marks
  natural  0.0000  0.5990  0.0705  0.0001 -0.0706 -0.5989  0.0000
  clamped  1.7989  0.1193  0.1904  0.0001 -0.1905 -0.1192 -1.7989

depth   table  natural   error  clamped   error
   10    90.03   108.93   18.90    75.95  -14.08
   30   442.21   437.17   -5.04   446.17    3.96
   50   892.09   893.73    1.64   890.73   -1.36
```

The natural spline has zero curvature at the ends by decree; the clamped spline finds 1.80, closer to the truth — the tank's true curvature is infinite there. Both have $M_3 \approx 0$ *at 60 cm, the inflection. The splines beat the straight line everywhere but lose to the degree-6 polynomial on this tank. Book: Section 7.6.*

Where the Splines Suffer

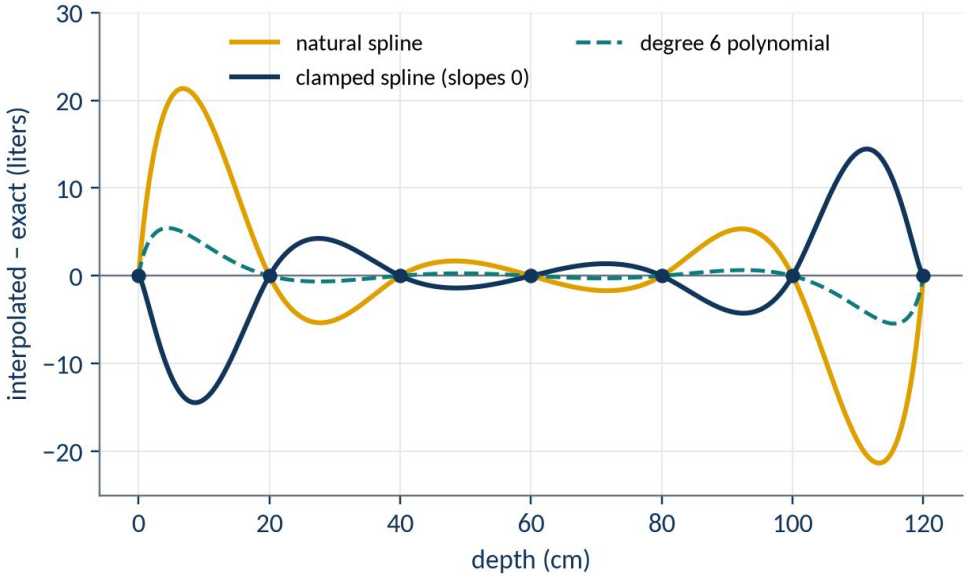


Figure 7.4 Error of the natural spline, the clamped spline (end slopes 0) and the degree-6 polynomial through the 20 cm marks. Near the ends, where the tank's curvature is infinite, the splines suffer most.

KEY INSIGHT. No interpolant is best everywhere. The tank's volume is very smooth in the middle and singular at the ends; the right answer is to measure. Denser marks help most: 10 cm marks cut the clamped spline's worst error from 14.45 L to 5.08 L.

Figure 7.4: the program also reads 47 cm as 821.4 L, exact to the litre. For Pak Darto the practical advice is simple: paint a mark every 10 cm, and use a curve, not a straight line, near the bottom and top. Book: Section 7.6.

Checking Against the Library

```
NumPy 2.5.3, SciPy 1.18.1
degree 6: Newton vs BarycentricInterpolator 9.1e-13
numpy.interp largest error 34.49 L
natural: ours vs CubicSpline 4.5e-13, largest error 21.34 L
clamped: ours vs CubicSpline 1.2e-12, largest error 14.45 L
not-a-knot (SciPy default): largest error 7.87 L
```

`BarycentricInterpolator` is the Lagrange polynomial in a numerically stable form; `numpy.interp` is the straight line; `CubicSpline` does all three end conditions. SciPy's default, not-a-knot, is the best spline here at 7.87 L. When the table comes from a known formula, as this one does, the best choice of all is to use the formula. Book: Sections 7.7–7.8.

Making It Better

Use what you know. When a table comes from a known formula, use the formula. Interpolation is for tables whose origin is unknown or too complicated to model.

Put the marks where the function bends. Seven marks at Chebyshev depths (1.5, 13.1, 34.0, 60.0, 86.0, 106.9 and 118.5 cm) give a degree-6 polynomial with a worst error of 3.95 L, against 5.45 L for equal spacing.

Monotone interpolation. Volumes only grow with depth, but a spline can dip between two points and make an extra centimetre of fuel "lose" litres. PCHIP chooses slopes so the interpolant is monotone wherever the data are.

Barycentric evaluation. The Lagrange form rewritten with weights computed once evaluates in $O(n)$ and is numerically stable; at Chebyshev points it is the method of choice for any degree.

Each of these improvements is one line in SciPy: `PchipInterpolator`, `BarycentricInterpolator`, *and a list of Chebyshev depths for the painter. Book: Section 7.8.*

Numerical Methods in Practice: Profitina

Profitina's historical analytics dashboard shows each brand's AI visibility score **week by week**, with a CSV export. Between two weekly scans there is no measurement.

A chart that draws a line between weekly points interpolates. A smooth spline can **overshoot**: a peak above the highest real score, a value that was never measured.

The honest choices are straight segments or a monotone, shape-preserving interpolant such as **PCHIP**, which never creates a maximum the data do not have.

The choice between a straight line, a polynomial and a spline is a choice about what the reader will be told happened in between. Book: Section 7.10.

Common Mistakes

Raising the degree to improve accuracy. With equal spacing it can be much worse (Runge). Use Chebyshev points, or splines.

Extrapolating. Outside the table the node polynomial grows rapidly, and so does the error.

Solving the Vandermonde system. Same polynomial, badly conditioned matrix; use Newton or barycentric.

Trusting natural end conditions. Clamp if the end slopes are known; otherwise not-a-knot.

Forgetting that splines can overshoot. For totals or percentages, use a monotone interpolant.

For your own tank, drum or odd-shaped jerry can: pour in a known volume at a time, a 20-litre can is convenient, record the depth after each pour, and measure more often near the bottom and the top. Book: Sections 7.9 and 7.11.

Summary

1. Straight lines (Equation (7.2)) are simple; their error follows the curvature: 34 L near the tank's ends, 3.4 L in the middle.

2. One polynomial of degree n passes through $n+1$ points: Lagrange (Equation (7.3)) or Newton with divided differences (Equation (7.4), Equation (7.5)).

3. Its error (Equation (7.7)) explodes near the ends for equal spacing (Runge); Chebyshev points (Equation (7.9)) fix it.

4. A cubic spline's curvatures solve a tridiagonal system (Equation (7.12)) in linear time (Equation (7.13)).

5. On this tank the degree-6 polynomial beats the splines; denser marks help most. Measure before you trust.

The Java and Python programs agree byte for byte, and NumPy and SciPy reproduce every interpolant to about 10^{-12}*. Book: Section 7.12.*

Next Time

Chapter 8 — Curve Fitting: Least Squares Regression. When the data are noisy, the curve should pass *near* the points, not through them.

Straight-line and polynomial regression, linearized models, and how to judge a fit.

Before then: from the wall chart, compute by hand the straight-line reading at 30 cm and the parabola through 20, 40 and 60 cm.

Compare both readings with the tank maker's 442.21 L; the solutions are in the book. Book: Sections 7.13–7.14.

Lecture 8 — Curve Fitting: Least Squares Regression

Course: Numerical Methods

Book: Irfan Subakti, *Numerical Methods: From Error Analysis to Machine Learning, with Java and Python*, Chapter 8

Lecturer: Irfan Subakti — yifana@gmail.com

Interpolation passes through every point, which is right when the points are exact. Measured data are not: they scatter. Today the curve passes near the points, chosen to make the squared misses as small as possible — least squares — and we learn to say how much the answer can be trusted and how to avoid fitting the noise. Book: Chapter 8, Sections 8.1–8.14.

What This Meeting Covers

1. The problem — an ojek driver's notebook of distance, trips, fuel, speed and battery health.

2. The straight line — the sum of squared errors, the normal equations, R^2 and standard errors.

3. More predictors — the design matrix, solved by QR from Chapter 6.

4. Curves that are linear in their coefficients — polynomials, and exponentials through logarithms.

5. Overfitting — judging a model on held-back data, and the library check.

Three files per language: shared helpers `lsq`, `line_fit` for lines and exponentials, `poly_fit` for polynomials. Java and Python print identical text. Book: Chapter 8.

Mas Dimas's Notebook

Mas Dimas drives an **ojek** through a ride-hailing app. Fuel is his biggest daily cost.

For forty days he wrote down the **distance**, the **trips**, and the **litres** bought to refill; some days the **average speed**; every two months his phone's **battery health**.

Four questions: What does one kilometre really cost in fuel? How much to plan for 120 km and 17 trips tomorrow? At what speed does the bike go furthest per litre? When does the battery reach **80%**?

None of the four answers can be read off the notebook directly: every day scatters around a trend, because of traffic, rain, waiting and hills. The trend is what we want, and the scatter tells us how sure to be. Book: Section 8.1.

Forty Days on One Plot

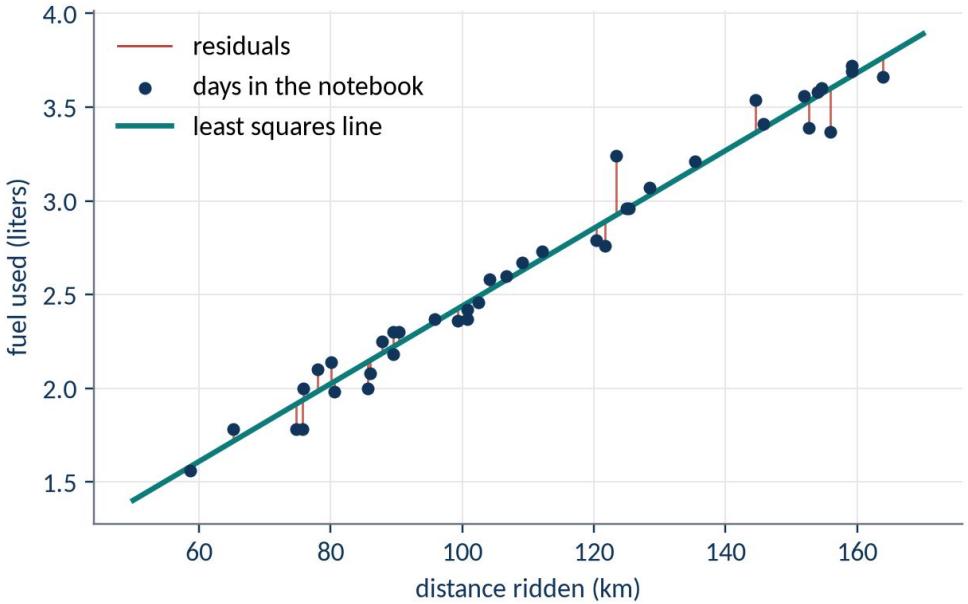


Figure 8.1 Forty days of Mas Dimas's notebook: fuel used against distance ridden, the least squares line, and the residuals, the vertical misses that the line makes as small as possible in the sum of squares.

Figure 8.1: the points lie close to a line, but not on it. A line through two chosen points would depend on which two; a line that listens to all forty is what we build. Book: Sections 8.1–8.2.

The Idea: Make the Squared Misses Small

For a line $a+b\,x$, day i misses by the **residual** $e_i = y_i - (a+b\,x_i)$. Choose a and b to minimize

$$\mathrm{SSE}(a,b) = \sum_{i=1}^{n} (y_i - a - b\,x_i)^2 \tag{8.1}$$

Why squares? Positive and negative misses cannot cancel, big misses count more, and the minimum is found by solving a *linear* system.

Equation (8.1): Legendre published it in 1805 for the orbits of comets, and Gauss used it for the orbits of planets. Minimizing the sum of absolute misses is also sensible, but it has no formula and needs the optimization methods of Chapter 13. Book: Section 8.2.

The Normal Equations and the Line

Set both partial derivatives of SSE to zero:

$$\begin{aligned} n\,a + (\sum x_i)b &= \sum y_i \\ (\sum x_i)a + (\sum x_i^2)b &= \sum x_i y_i \end{aligned} \tag{8.2}$$

Solved in the stable, centred form:

$$b = \frac{S_{xy}}{S_{xx}},\; a = \acute{y} - b\,\acute{x},\; S_{xx} = \sum_i (x_i - \acute{x})^2,\; S_{xy} = \sum_i (x_i - \acute{x})(y_i - \acute{y}) \tag{8.3}$$

Equation (8.2) is the 2×2 *case of Chapter 6's normal equations. Equation (8.3) subtracts the means first, which avoids the cancellation of Chapter 1, and shows that the line always passes through the point of means* $(\acute{x}, \acute{y})$*. Book: Section 8.3.1.*

By Hand: the First Five Days

Table 8.1 The least squares line through the first five days, by hand: deviations from the means, their squares and their products.

Day	x_i (km)	y_i (L)	$x_i - \bar{x}$	$y_i - \bar{y}$	$(x_i - \bar{x})^2$	Product
1	74.7	1.78	−32.86	−0.812	1079.78	26.6823
2	95.8	2.37	−11.76	−0.222	138.30	2.6107
3	154.6	3.60	47.04	1.008	2212.76	47.4163
4	87.8	2.25	−19.76	−0.342	390.46	6.7579
5	124.9	2.96	17.34	0.368	300.68	6.3811
sum	537.8	12.96	0	0	4121.97	89.848

Table 8.1: $\bar{x} = 107.56$, $\bar{y} = 2.592$, *so* $b = 89.848 / 4121.97 = 0.0218$ *L/km and* $a = 2.592 - 0.0218 \times 107.56 = 0.248$ *L. Five days give a first estimate; forty days give a better one. Notice that the deviation columns add up to zero, as they must, and that the whole computation needs only sums, one pass through the notebook after the means are known. Book: Section 8.2.*

How Good Is the Line?

The fraction of the variation explained, with SST the sum of squares of $y_i - \acute{y}$:

$$R^2 = 1 - \frac{\text{SSE}}{\text{SST}} \tag{8.4}$$

The typical miss, and the uncertainty of the slope:

$$s_e = \sqrt{\frac{\text{SSE}}{n-2}} \tag{8.5}$$

$$\text{SE}(b) = \frac{s_e}{\sqrt{S_{xx}}} \tag{8.6}$$

Equation (8.4): 1 means a perfect fit, 0 means the line explains nothing. Equation (8.5) divides by $n-2$ because two numbers were estimated from the data. Equation (8.6): the slope is better known when the x values are spread wide — ride on both short and long days. About two standard errors either side gives a 95% range. Book: Section 8.3.2.

Predicting One New Day

$$\hat{y}_0 \pm t\, s_e \sqrt{1 + \frac{1}{n} + \frac{(x_0 - \acute{x})^2}{S_{xx}}} \tag{8.7}$$

The “1” is the new day’s own scatter; the other two terms are the line’s uncertainty, growing away from $\acute{x}$.

For 150 km: **3.48 ± 0.21 L**. Plan for the upper end, and the bike will not run dry.

Equation (8.7) is a prediction interval, wider than the uncertainty of the line itself, because a single day scatters. Far from the data the last term grows: a prediction for 400 km would be both uncertain and outside the range the model has seen. Book: Section 8.3.3.

More Predictors: the Design Matrix

Trips also cost fuel: waiting, starting, detours. Model $y \approx \beta_0 + \beta_1 x_1 + \beta_2 x_2$, one row per day:

$$y \approx X\beta \tag{8.8}$$

$$X^T X \beta = X^T y \tag{8.9}$$

*Equation (8.8): the **design matrix** X has a column of ones, a column of kilometres and a column of trips. Equation (8.9) are the normal equations of Chapter 6; we solve with Householder QR, never forming $X^T X$. Book: Section 8.4.*

Running It: Lines and Predictors

```
40 days, mean 111.75 km and 2.6825 L per day
Sxx = 36092.60, Sxy = 748.4575
liters = 0.3652 + 0.020737 * km
R^2 = 0.9757, standard error 0.1009 L
slope 0.020737 +/- 0.001062 L/km (about 95%)
largest residual: day 17, +0.316 L

liters = 0.2654 + 0.018568 * km + 0.02285 * trips
R^2 = 0.9894, standard error 0.0675 L
tomorrow, 120 km and 17 trips: 2.88 L
```

`line_fit.py` and `LineFit.java`. *Day 17 used 0.316 L more than predicted, three standard errors — worth asking why before deleting it. With trips, R^2 rises to 0.9894 and the typical miss falls to 0.0675 L. The trip coefficient, 0.023 L per trip, is the effect at a fixed distance. A 10 km ride costs about 0.2 L; comparing that with the fare is Mas Dimas's decision. Book: Section 8.8.*

Curves That Are Linear in Their Coefficients

A polynomial is linear in its **coefficients**, even though it is curved in x:

$$y \approx \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_d x^d \tag{8.10}$$

Fuel economy against speed is a hill. With the scaled speed $t = (\text{speed} - 45)/25$ and a parabola $c_0 + c_1 t + c_2 t^2$, the top is at

$$t^* = -\frac{c_1}{2c_2}, \quad \text{speed}^* = 45 + 25\,t^* \tag{8.11}$$

Equation (8.10): each power is one more column of $\boldsymbol{X}$. Centre and scale first: powers of 70 up to the eighth are 15 digits long and nearly parallel columns. Equation (8.11) is the vertex of the parabola in the scaled variable, converted back to km/h. Book: Sections 8.5.1–8.5.2.

Exponentials Through Logarithms

A battery loses capacity roughly in proportion to what it has left: $H(t) = C\,e^{kt}$, not linear in C and k. Its logarithm is:

$$\ln H = \ln C + k\,t \tag{8.12}$$

Fit a straight line to $(t_i, \ln H_i)$, transform back, and solve $C\,e^{kt} = 80$ for t.

Equation (8.12): a straight line with intercept $\ln C$ and slope k. Fitting $\ln y$ minimizes relative errors, not absolute ones; for precise answers in the original units, check with a direct nonlinear fit, as the library comparison does. Book: Section 8.5.3.

The Battery Curve

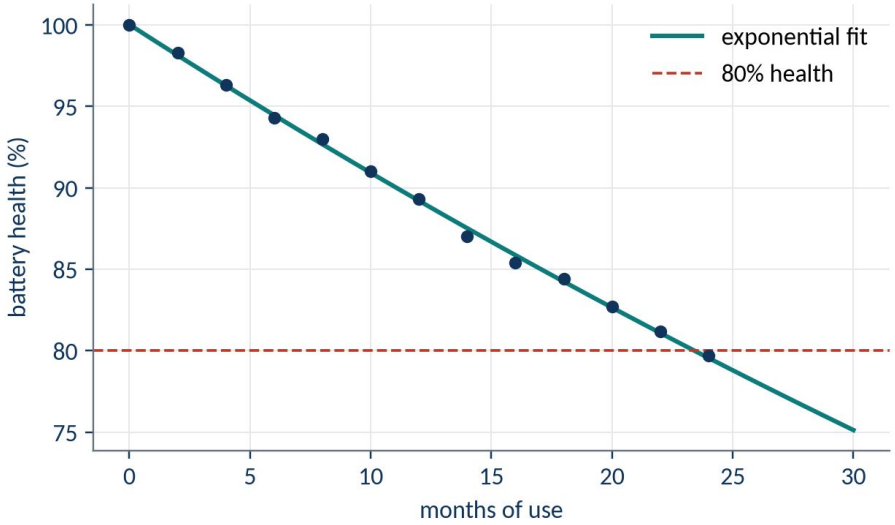


Figure 8.2 Battery health of Mas Dimas's phone every two months, the exponential fit through logarithms, and the 80% level it reaches after about 23.4 months.

```
battery health = 100.03 * exp(-0.00954 * month)
80% is reached after about 23.4 months
```

Figure 8.2: over two years an exponential with a small rate looks almost straight; a straight line fitted directly would say 23.1 months. The difference is small here, but the exponential keeps making sense beyond the data, where the line would eventually reach zero. Book: Section 8.8.

Overfitting: Higher Degree, Better Fit, Worse Model

Adding a column can only lower SSE, so R^2 rises with every degree. A degree-35 polynomial through 36 points reaches $R^2 = 1$ and predicts nothing.

Fit on 24 days, **hold back 12**, and judge on the held-back days:

$$\text{RMSE} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2} \tag{8.13}$$

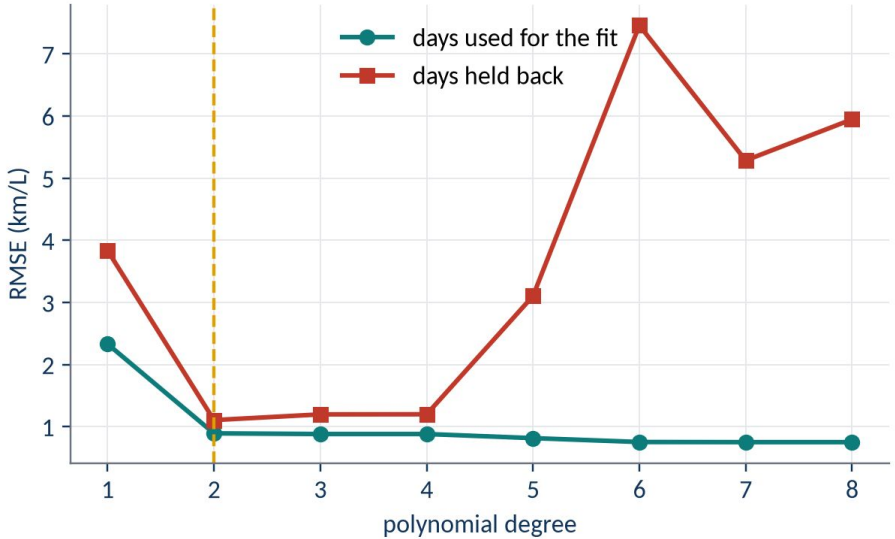


Figure 8.3 RMSE of polynomial fits of degree 1 to 8 on the 24 fitting days and on the 12 held-back days. The fitting error only falls; the held-back error is smallest at degree 2 and grows sharply from degree 5.

Equation (8.13) on data the model has not seen is the honest judge. Figure 8.3: the classic U shape of held-back error — too simple on the left, too flexible on the right. Chapter 14 meets the same idea as the validation set of machine learning. Book: Section 8.6.

Running It: Choosing the Degree

```
36 days: 24 to fit, 12 held back
degree  R^2 (fit)  RMSE fit  RMSE held back
     1     0.3998     2.330          3.834
     2     0.9120     0.892          1.106
     3     0.9141     0.881          1.198
     4     0.9141     0.881          1.199
     5     0.9265     0.815          3.103
     6     0.9373     0.753          7.460
     7     0.9377     0.751          5.286
     8     0.9377     0.751          5.948

all days, degree 2: km/L = 54.653 +0.535 x -8.459 x^2
  where x = (speed - 45) / 25
best average speed 45.8 km/h, about 54.7 km/L
```

*`poly_fit.py` and `PolyFit.java`. The line fits badly on both groups because the data are curved. Degree 2 drops the held-back error to 1.106; degrees 3 and 4 fit the training days a hair better and the held-back days worse; from degree 5 the held-back error explodes. Degree 2 wins, and its vertex puts the best average speed at **45.8 km/h**. Book: Section 8.8.*

The Speed Curve

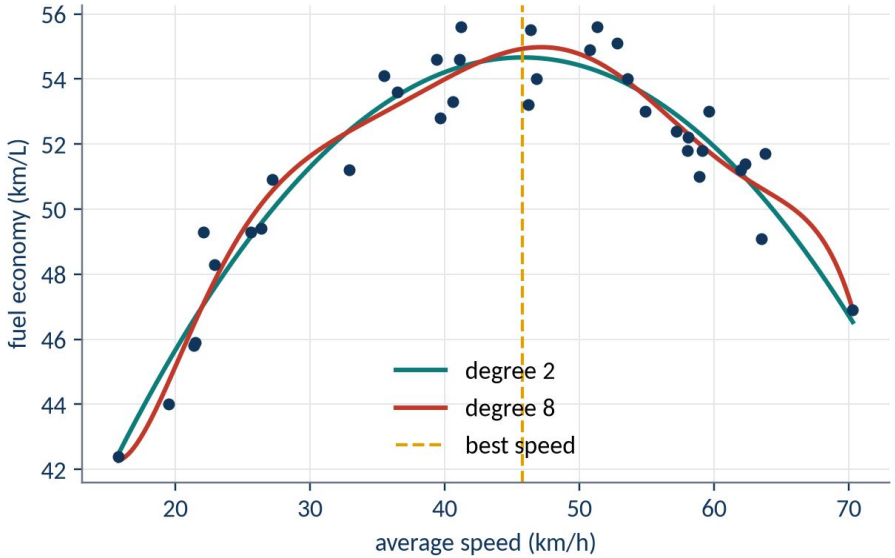


Figure 8.4 Fuel economy against average speed on 36 days, with the degree-2 fit (whose peak at 45.8 km/h is marked) and the degree-8 fit, which follows individual days instead of the trend.

Figure 8.4: the degree-8 curve chases individual days and wiggles where no day was measured; the parabola tells the story a rider can use. Book: Section 8.8.

Checking Against the Library

```
NumPy 2.5.3, SciPy 1.18.1
linregress: intercept 0.3652, slope 0.020737, R^2 0.9757
  standard error of the slope 0.000531
lstsq with trips: [0.265357 0.018568 0.022854]
Polynomial.fit degree 2: best speed 45.8 km/h, 54.7 km/L
curve_fit: health = 100.04 * exp(-0.00956 * month), 80% after 23.4 months
```

`linregress` reports the slope's standard error, 0.000531; our "±0.001062" is two of them, the 95% range. `lstsq` uses the singular value decomposition, the most robust member of the QR family. `curve_fit` fits the exponential directly in percent and gives almost the same curve and the same 23.4 months. Book: Section 8.9.

Numerical Methods in Practice: Profitina

Profitina's historical analytics dashboard keeps a **weekly AI visibility score** for each brand, with alerts and a CSV export.

"Is the site getting better?" is answered honestly by the **slope** of a least squares line through the weeks, with its standard error Equation (8.6).

$\nearrow 0.8 \pm 0.3$ points per week: improving, and not by accident. $\nearrow 0.8 \pm 1.5$: cannot tell yet, however encouraging the last two weeks look.

Twelve weeks give a slope known to about ± 0.25 points per week; twenty-six weeks give a standard error more than three times smaller. Patience buys certainty, which is worth saying plainly to a brand owner who wants a verdict after three weeks. The numbers advise; the people decide. Book: Section 8.12.

Common Mistakes

Judging a model by R^2 alone. It never decreases when a column is added; compare on held-back data.

High-degree polynomials in raw units. Centre and scale, and solve by QR.

Minimizing vertical misses when the error is in x. Here the distance is accurate and the fuel is not, so vertical is right.

Extrapolating a fit. A model is trustworthy only over the range of its data.

Throwing away the "outlier" without asking why. Day 17 may be a mistake, or the only day with a long wait in the rain.

The same routine fits sales against temperature for an ice-tea stall, electricity against air-conditioner hours, or water use against guests at a small inn: put the columns in a CSV file, plot first, fit, look at the residuals, and hold some data back. Book: Sections 8.11 and 8.13.

Summary

1. Least squares minimizes the sum of squared residuals (Equation (8.1)); for a line, slope S_{xy}/S_{xx} through the means (Equation (8.3)).

2. R^2 (Equation (8.4)), the standard error (Equation (8.5)) and the slope's standard error (Equation (8.6)) say how much to trust it: 0.0207 ± 0.0011 L/km.

3. A new day is predicted with a wider interval (Equation (8.7)): 3.48 ± 0.21 L for 150 km.

4. Several predictors enter through the design matrix (Equation (8.8)), solved by QR; polynomials and logged exponentials are linear least squares too.

5. Only held-back error (Equation (8.13)) reveals overfitting; here degree 2 wins, best speed 45.8 km/h, battery at 80% after 23.4 months.

The Java and Python programs agree byte for byte, and `linregress`, `lstsq`, `Polynomial.fit` *and* `curve_fit` *confirm every result. Book: Section 8.14.*

Next Time

Chapter 9 — Numerical Differentiation. Speed and acceleration from positions measured at intervals.

Forward, backward and central differences, their errors, Richardson extrapolation — and what noise does to a derivative.

Before then: using the five-day table, check that the residuals add up to zero and compute the five-day R^2.

The five values of $y_i - \hat{y}$ give SST; the solution is in the book, together with weighted least squares for days of different reliability. Book: Sections 8.15–8.16.

Lecture 9 — Numerical Differentiation

Course: Numerical Methods

Book: Irfan Subakti, *Numerical Methods: From Error Analysis to Machine Learning, with Java and Python*, Chapter 9

Lecturer: Irfan Subakti — yifana@gmail.com

A derivative is a rate of change: speed from distance, acceleration from speed, growth from a score. When we have only samples, we replace the tangent by a secant. Today we derive the formulas and their errors from Taylor's theorem, meet the step-size dilemma again, sharpen estimates with Richardson extrapolation, and learn what noisy data do to a derivative. Book: Chapter 9, Sections 9.1–9.12.

What This Meeting Covers

1. The problem — an app says “harsh braking detected”. Was it right?

2. The formulas — forward, backward, central, second differences, from Taylor’s theorem.

3. Two kinds of error — truncation against round-off, and the optimal step.

4. Richardson extrapolation — combining two steps to cancel the leading error.

5. Noisy data — the trip analyzer, a false alarm, and smoothing before differentiating.

The trip analyzer exists in Java and Python and prints the same table; NumPy’s `np.gradient` reproduces its speeds exactly. Book: Chapter 9.

Mas Dimas's Braking Report

Mas Dimas drives an online ojek in **Yogyakarta**, short trips around Jalan Kaliurang.

One evening the app says: ***harsh braking detected***. A few warnings, and a driver's safety score goes down, and with it the better orders.

He remembers braking hard for a cat at about 40 km/h — right, and he would do it again. But on another trip the app complained when he only rolled gently to a stop at a customer's gate.

The phone logs only **distance, once a second**. How does it know how hard he braked?

Speed is the first derivative of distance, acceleration the second. The phone must estimate both from samples — and the samples are noisy. Book: Section 9.1.

The Idea: Secants Instead of Tangents

Table 9.1 Five distance readings from Mas Dimas's trip, one per second, around the hard brake.

Time t (s)	16	17	18	19	20
distance s (m)	148.2	159.5	168.7	174.4	176.5

Speed at 18 s from the neighbours on both sides: $(174.4 - 159.5)/2 = 7.45$ m/s.

Acceleration from three readings, the **second difference**: $(174.4 - 2\times 168.7 + 159.5)/1^2 = -3.5$ m/s^2.

Table 9.1: a deceleration of 3.5 m/s^2 means losing 12.6 km/h of speed every second — a real hard brake. The question is how accurate such numbers are, and that needs Taylor's theorem. Book: Section 9.2.

Deriving the Formulas from Taylor's Theorem

$$f(x+h)=f(x)+hf'(x)+\frac{h^2}{2}f''(x)+\frac{h^3}{6}f'''(x)+\cdots+\frac{h^n}{n!}f^{(n)}(x)+R_n \tag{9.1}$$

Stop after the first derivative and solve for it — the **forward difference**, first order:

$$f'(x)\approx\frac{f(x+h)-f(x)}{h},\text{error}=-\frac{h}{2}f''(\xi) \tag{9.2}$$

Equation (9.1) is the tool behind nearly every error estimate of the course. Equation (9.2): halve h and the error halves. The backward difference is the mirror image, with the opposite sign of error. Book: Sections 9.3.1–9.3.2.

The Central Difference and the Second Derivative

Subtract the expansions of $f(x+h)$ and $f(x-h)$: the h^2 terms cancel.

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}, \text{error} = -\frac{h^2}{6} f'''(\xi) \tag{9.3}$$

Add them instead: the odd terms cancel.

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}, \text{error} = -\frac{h^2}{12} f^{(4)}(\xi) \tag{9.4}$$

Equation (9.3) is second order: halve h and the error falls by four, for the same two function values. Equation (9.4) is second order too, and exact for cubics. Notice the h^2 in its denominator: a reading error in each value is magnified by $1/h^2$. Book: Sections 9.3.3–9.3.4.

Formulas for the Ends

At the first reading there is no left neighbour. Choose coefficients so that the Taylor terms cancel up to h^2 — **undetermined coefficients**:

$$f'(x) \approx \frac{-3f(x)+4f(x+h)-f(x+2h)}{2h}, \text{error} = \frac{h^2}{3} f'''(\xi) \tag{9.5}$$

Table 9.2 Finite-difference formulas for the first and second derivative, with the number of function values they need and their truncation error (ξ omitted).

Derivative	Formula	Points	Truncation error
f' forward	$[f(x+h)-f(x)]/h$	2	$-\frac{h}{2}f''$
f' central	$[f(x+h)-f(x-h)]/(2h)$	2	$-\frac{h^2}{6}f'''$
f' three-point forward	$[-3f(x)+4f(x+h)-f(x+2h)]/(2h)$	3	$+\frac{h^2}{3}f'''$
f' five-point central	$[f(x-2h)-8f(x-h)+8f(x+h)-f(x+2h$	4	$+\frac{h^4}{30}f^{(5)}$
f'' central	$[f(x+h)-2f(x)+f(x-h)]/h^2$	3	$-\frac{h^2}{12}f^{(4)}$

Equation (9.5) keeps second order at the ends, so the whole trip table has the same accuracy. Table 9.2 collects the common formulas; the book also lists the four-point one-sided second derivative used at the ends. Book: Section 9.3.5.

Two Kinds of Error, Again

```
     h     forward        error    central        error
 1e-01  0.2484567313  1.54e-03  0.2500195366  1.95e-05
 1e-02  0.2498439450  1.56e-04  0.2500001953  1.95e-07
 1e-03  0.2499843770  1.56e-05  0.2500000020  1.95e-09
 1e-04  0.2499984375  1.56e-06  0.2500000000  2.05e-11
 1e-05  0.2499998438  1.56e-07  0.2500000000  1.64e-12
 1e-06  0.2499999843  1.57e-08  0.2499999999  7.61e-11
```

`diff_steps.py` and `DiffSteps.java` differentiate $\sqrt{x}$ at 4 (true value 0.25). The forward error falls tenfold per row, the central error a hundredfold — until $h = 10^{-5}$. Below that the central difference gets worse: round-off, about $2\varepsilon f(x)/h$, has taken over. Book: Section 9.4.

The Optimal Step

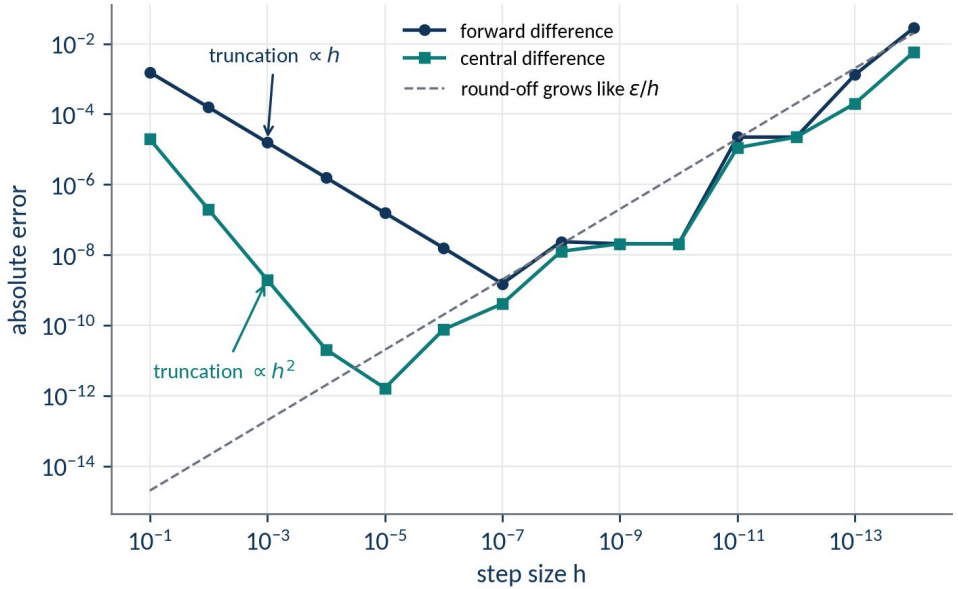


Figure 9.1 Error of the forward and central differences of $\sqrt{x}$ at $x = 4$ against the step h (both axes logarithmic, h shrinking to the right). Truncation error falls like h or h^2; round-off error rises like ε / h.

$$h^{*} = 2\sqrt{\frac{\varepsilon|f|}{M_2}} \text{(forward)}, h^{*} = \left(\frac{3\varepsilon|f|}{M_3}\right)^{1/3} \text{(central)} \tag{9.6}$$

Figure 9.1: two V shapes, the central one lower and further left. Equation (9.6): about 10^{-8} for the forward and 10^{-5} for the central difference in double precision. For measured data the "round-off" is the reading error — for Mas Dimas's phone around 0.1 to 0.3 m — so the same argument says: do not trust tiny steps. Book: Section 9.4.

Richardson Extrapolation

The central difference has an error in even powers of h: $D(h) = f'(x) + a_1 h^2 + a_2 h^4 + \cdots$. Then $\dfrac{4D(h/2) - D(h)}{3}$ cancels the h^2 term. Repeat:

$$D_{i,j} = \frac{4^j D_{i,j-1} - D_{i-1,j-1}}{4^j - 1}, \quad j = 1, 2, \ldots, i \tag{9.7}$$

```
h = 0.800  0.251272405013
h = 0.400  0.250313875299  0.249994365394
h = 0.200  0.250078210575  0.249999655667  0.250000008352
h = 0.100  0.250019536593  0.249999978598  0.250000000127  0.249999999996
```

Equation (9.7): each column removes one more error term. With steps no smaller than 0.1 — far above the round-off danger zone — the diagonal reaches ten correct digits: error 1.27×10^{-3} at level 0, 5.63×10^{-6} at level 1, and so on. The same idea drives Romberg integration in Chapter 10. Book: Section 9.5.

The Trip Analyzer

Listing 9.1 From `speed.py`: acceleration by the central second difference, with one-sided formulas at the ends.

```
def accelerations(s, h):
    """Second derivative: central inside, four-point one-sided at the ends."""
    n = len(s)
    a = [0.0] * n
    a[0] = (2 * s[0] - 5 * s[1] + 4 * s[2] - s[3]) / (h * h)
    for i in range(1, n - 1):
        a[i] = (s[i + 1] - 2 * s[i] + s[i - 1]) / (h * h)
    a[n - 1] = (2 * s[n - 1] - 5 * s[n - 2] + 4 * s[n - 3] - s[n - 4]) / (h * h)
    return a
```

Listing 9.1: the speeds use Equation (9.3) inside and Equation (9.5) at the ends; braking harder than -3.0 *m/s² is reported. The log* `data/trip.csv` *was prepared to match a trip of exactly this kind, so we know what really happened: about 40 km/h, the cat, a slower stretch, and a gentle stop. Book: Section 9.6.*

Running It: Three Alarms

```
  t     s(m)   v(m/s)  v(km/h)  a(m/s^2)
  0      0.0    -0.90     -3.2      3.90
  1      0.6     2.10      7.6      3.00
 ...
 17    159.5    10.25     36.9     -2.10
 18    168.7     7.45     26.8     -3.50  harsh braking
 19    174.4     3.90     14.0     -3.60  harsh braking
 20    176.5     2.15      7.7      0.10
 ...
 35    287.9     6.85     24.7     -3.10  harsh braking
 36    293.2     5.35     19.3      0.10
top speed 41.4 km/h at t = 11 s
```

*Rows shown from the full 41-line output (the dots mark rows left out here). A **negative starting speed**, a top speed of 41.4 km/h the bike never reached, acceleration swinging between* -2.3 *and* $+1.6$ *m/s² at steady cruise, and a **false alarm at 35 s** — the gentle stop. The formulas are right; the readings are noisy. Book: Section 9.6.*

Why Noise Hurts Derivatives

KEY INSIGHT. Differentiation amplifies noise. With independent reading errors of standard deviation σ, the speed carries noise $\sqrt{2}\sigma/(2h)$ and the acceleration $\sqrt{6}\sigma/h^2$. For $\sigma = 0.3$ m and $h = 1$ s that is 0.73 m/s² — enough to push a gentle stop past the alarm.

A smaller h is not the cure: it makes $1/h^2$ larger. The cure is to use **more readings** per estimate.

The variance of $s_{i+1} - 2s_i + s_{i-1}$ is $(1+4+1)\sigma^2$, hence the $\sqrt{6}$. Integration, next chapter, is the opposite: it averages noise away. Book: Section 9.6.

Smooth Before You Differentiate

Fit a least-squares parabola (Chapter 8) through **five** readings and differentiate the parabola:

$$v \approx \frac{-2s_{-2} - s_{-1} + s_1 + 2s_2}{10h}, a \approx \frac{2s_{-2} - s_{-1} - 2s_0 - s_1 + 2s_2}{7h^2} \tag{9.8}$$

```
 17    10.25      9.43     -2.10     -1.73
 18     7.45      7.15     -3.50     -3.13
 19     3.90      4.62     -3.60     -2.51
 20     2.15      3.05      0.10     -0.73
 ...
 35     6.85      6.77     -3.10     -1.16
harsh braking, plain formulas:  t = 18, 19, 35
harsh braking, smoothed:        t = 18
```

Equation (9.8) are Savitzky–Golay-type formulas. Columns: time, plain and smoothed speed, plain and smoothed acceleration. The hard brake is still found at 18 s with -3.13 *m/s², a little milder than the true* -3.5 *because the fit spreads it; the false alarm at 35 s becomes* -1.16*, close to the true* -1.4*. At steady cruise the smoothed acceleration stays within* ± 0.4*. Book: Section 9.7.*

The Trip on One Plot

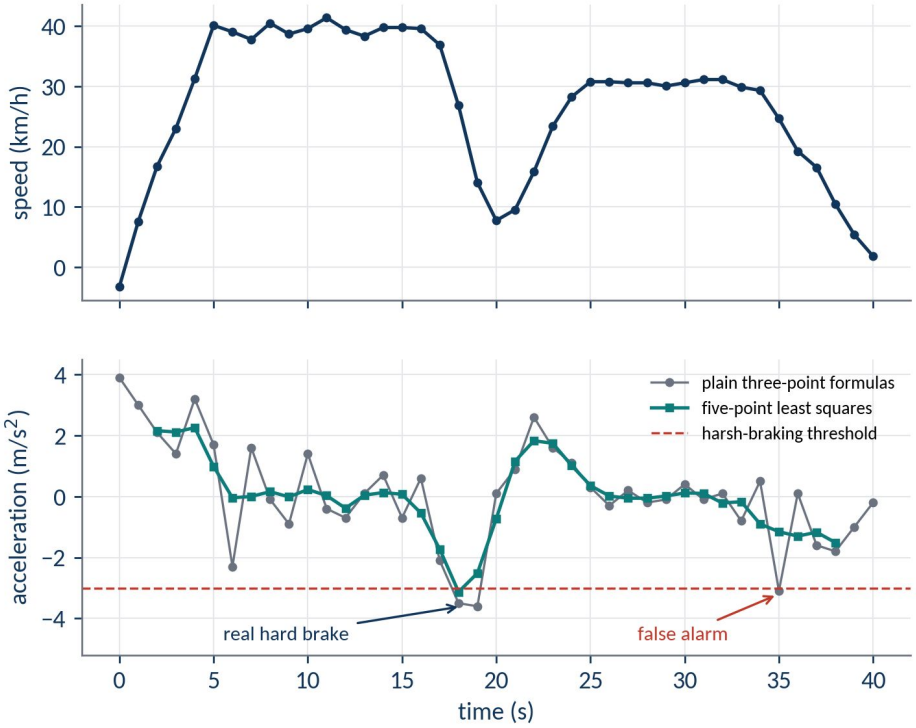


Figure 9.2 Speed of Mas Dimas's trip (top) and its acceleration (bottom) from the plain three-point formulas and from the five-point least-squares formulas; only the smoothed curve separates the real hard brake from the false alarm.

> **WARNING.** A smoothing formula uses readings from the future: at $t = 18$ it needs the readings at 19 and 20. An app that must warn *while* the driver brakes can only use one-sided formulas, which are noisier, or accept a short delay. Every real-time derivative faces this choice between noise and delay.

Figure 9.2: the plain acceleration wanders across the red threshold; the smoothed one crosses it only where the brake was. Mas Dimas was right about both trips. Book: Section 9.7.

Checking Against the Library

```
NumPy 2.4.4, SciPy 1.17.1
largest speed difference, ours vs np.gradient: 0.0e+00 m/s
a at t = 19 s: ours -3.60, gradient twice -2.65
scipy derivative of sqrt at 4: 0.250000000000000 (estimated error 8.9e-14, 11 evaluations)
```

`np.gradient` with `edge_order=2` uses the same central and one-sided formulas and reproduces our speeds exactly. Applying it twice for acceleration is not the second difference — it spans four seconds, not two, and gives -2.65 *instead of* -3.60 *at 19 s. `scipy.differentiate.derivative` differentiates a function adaptively, with an error estimate. The tests `test_diff.py` and `DiffTest.java` check facts that must hold whatever the data: the central difference is exact for a parabola, the forward error halves when* h *halves, and for* $s = t^2$ *the analyzer returns exactly* $v = 2t$ *and* $a = 2$*. Book: Section 9.8.*

Numerical Methods in Practice: Profitina

Profitina's historical analytics dashboard re-scans each brand's site on a schedule, stores a score per week, and raises **alerts**.

Is my score improving? How fast? Did it just drop? are derivative questions about a weekly series.

A single scan is noisy (a page that responded slowly), so an alert from two single weeks would give false alarms of exactly the kind that annoyed Mas Dimas.

The class of solution is Section 9.7's: estimate the trend from a least-squares fit over several recent weeks, accept that a real change is recognized a little later, and explain it in plain words. How Profitina computes its alerts is its own business; the shape of the problem is this chapter's. The numbers advise; the user decides. Book: Section 9.9.

Common Mistakes

Making h as small as possible. Below the optimum round-off takes over; use 10^{-8} forward, 10^{-5} central, or Richardson.

Differentiating raw measured data. Know your noise first; smooth with more points.

Applying a first-derivative routine twice for the second derivative. It widens the stencil and changes the answer.

Forgetting the ends. Central formulas need both neighbours; use the one-sided formulas of Table 9.2.

Before differentiating any log, repeat a measurement that should not change — a parked motorbike, a full tank — and see how the readings wobble. Book: Sections 9.10–9.11.

Summary

1. From Taylor (Equation (9.1)): forward is first order (Equation (9.2)), central and the second difference are second order (Equation (9.3), Equation (9.4)).

2. Undetermined coefficients give formulas for the ends (Equation (9.5)).

3. Truncation falls with h, round-off grows like $1/h$; the optimum (Equation (9.6)) is about 10^{-8} forward, 10^{-5} central.

4. Richardson (Equation (9.7)) reaches high accuracy with safe, large steps.

5. Measured noise is amplified by $1/h$ and $1/h^2$; the five-point least-squares formulas (Equation (9.8)) kept the real brake and removed the false alarm.

`np.gradient` with `edge_order=2` reproduces our speeds exactly; `scipy.differentiate.derivative` differentiates functions adaptively. Book: Section 9.12.

Next Time

Chapter 10 — Numerical Integration: Trapezoid, Simpson, Romberg and Gauss. From rates back to totals.

Area under a curve from samples, error orders, Richardson again as Romberg, and Gauss's clever choice of points.

Before then: from the readings at 34, 35 and 36 s, compute the plain speed and acceleration at 35 s by hand, and explain the false alarm with reading errors.

The exercise gives the true distances too, so you can compute each reading's error and see exactly how a gentle stop became -3.1 *m/s². Unevenly spaced readings are handled in the solutions. Book: Sections 9.13–9.14.*

Lecture 10 — Numerical Integration: Trapezoid, Simpson, Romberg and Gauss

Course: Numerical Methods

Book: Irfan Subakti, *Numerical Methods: From Error Analysis to Machine Learning, with Java and Python*, Chapter 10

Lecturer: Irfan Subakti — yifana@gmail.com

Adding up many small pieces is one of the oldest things people do with numbers: the area of a field from strips, the rain of a storm from readings every ten minutes, the energy a machine used. Each is an integral. Today we build the classic rules, prove their error orders, estimate errors without knowing the truth, and meet two smarter ideas: Romberg's extrapolation and Gauss's choice of points. Book: Chapter 10, Sections 10.1–10.14.

What This Meeting Covers

1. The problem — the area of an orchard between a straight road and a wandering river.

2. Trapezoid and Simpson — derivations, error orders, and an error estimate from halving.

3. Romberg — Richardson extrapolation of the trapezoidal rule.

4. Gauss quadrature — choosing where to measure, and adaptive Simpson.

5. Checking — the orders measured, and SciPy's current integration tools.

Differentiation amplified noise; integration averages it away. That is why integration is the friendlier of the two, and why its rules can be pushed to very high accuracy. Book: Chapter 10.

How Big Is the Orchard?

Pak Harjono's family owns a **fruit and durian orchard**: a strip between a straight village road and a small river, **160 m** along the road.

Anto, the son taking over, needs the area: durian trees stand about 10 m apart, so how many seedlings, and how much compost? And the old land letter gives an area nobody quite believes.

With a 50 m tape he measures, every 10 m, the distance at right angles from the road to the bank — the **offset**: 38.6 m at the start, 31.9 m at the far corner, 17 offsets in all.

Before paying a surveyor, Anto wants to know roughly what the answer should be. The official measurement is the surveyor's; the calculation is for planning. Book: Section 10.1.

The Idea: Strips

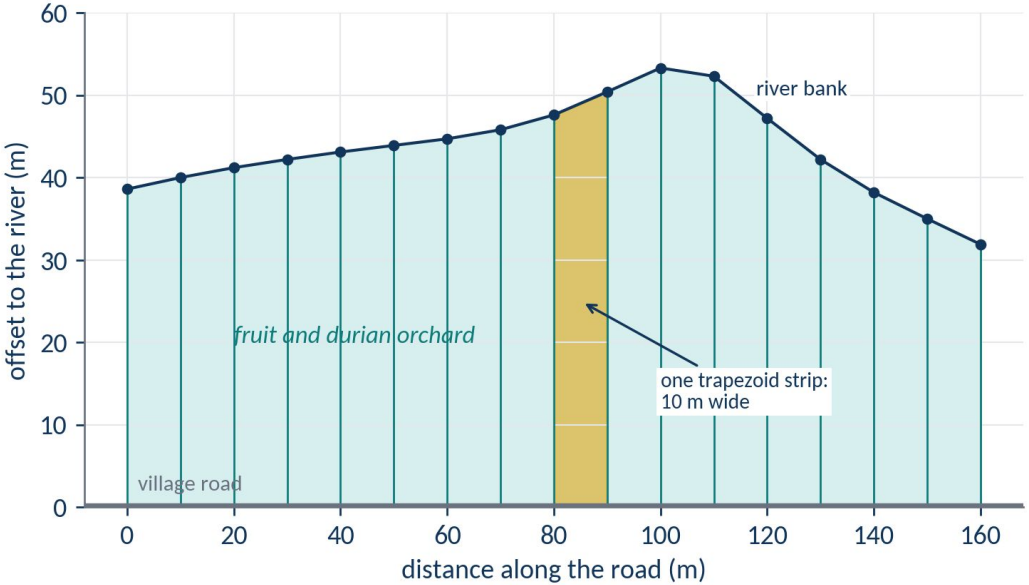


Figure 10.1 The orchard between the straight village road and the river bank, with the seventeen offsets measured every 10 m; the highlighted strip is one of the sixteen trapezoids of the trapezoidal rule.

Pretend each piece of bank is straight: each strip is a trapezoid. The first: $10 \times (38.6 + 40.0)/2 = 393.0\ \text{m}^2$.

Or fit a parabola through three offsets: the first two strips give $\frac{10}{3}\left(38.6+4\times 40.0+41.2\right)=799.3\ \text{m}^2$.

Figure 10.1: sixteen strips, each 10 m wide. Adding all trapezoids gives each inner offset full weight and the two end offsets half weight. The parabola version is Simpson's rule. Book: Section 10.2.

The Trapezoidal Rule and Its Error

One strip, from Taylor around the midpoint:

$$\int_a^b f(x)\,dx - \frac{h}{2}(f(a)+f(b)) = \left(\frac{1}{24} - \frac{1}{8}\right)h^3 f''(m) + \cdots = -\frac{h^3}{12}f''(\xi) \tag{10.1}$$

All n strips, $n=(b-a)/h$:

$$T(h) = h\left(\frac{f_0}{2} + f_1 + f_2 + \cdots + f_{n-1} + \frac{f_n}{2}\right), \quad I - T(h) = -\frac{(b-a)h^2}{12}f''(\xi) \tag{10.2}$$

Equation (10.1): each strip errs by h^3; there are $n \propto 1/h$ strips, so Equation (10.2) is second order. The sign follows the curvature: where the bank bulges outward, the straight tops cut off area and the trapezoids come out short. Book: Section 10.3.1.

The Euler–Maclaurin Formula

$$T(h) - I = \frac{h^2}{12}\left(f'(b) - f'(a)\right) - \frac{h^4}{720}\left(f'''(b) - f'''(a)\right) + \cdots \tag{10.3}$$

The error is a series in **even powers of** h, with coefficients that do not depend on h.

That is exactly the situation of Richardson extrapolation in Chapter 9.

Equation (10.3) has a surprising consequence: for a smooth periodic function integrated over a full period, all the derivative differences vanish and the trapezoidal rule converges faster than any power of h. It also underlies Simpson and Romberg, which follow. Book: Section 10.3.2.

Simpson's Rules

Weights for three points that integrate 1, x and x^2 exactly give $\frac{1}{3}, \frac{4}{3}, \frac{1}{3}$ — and, by symmetry, x^3 too:

$$S(h)=\frac{h}{3}(f_0+4f_1+2f_2+4f_3+\cdots+2f_{n-2}+4f_{n-1}+f_n), I-S(h)=-\frac{(b-a)h^4}{180}f^{(4)}(\xi) \tag{10.4}$$

$$S(h)=\frac{4T(h)-T(2h)}{3} \tag{10.5}$$

Equation (10.4) needs an even number of strips and is fourth order. Equation (10.5): Simpson's rule is one Richardson step on the trapezoidal rule. For an odd number of strips, use the 3/8 rule on the last three. Book: Sections 10.4.1–10.4.2.

Simpson's 3/8 Rule, and an Error Estimate for Free

$$\int_{x_0}^{x_3} f(x)\,dx \approx \frac{3h}{8}(f_0+3f_1+3f_2+f_3),\ \text{error} = -\frac{3h^5}{80}f^{(4)}(\xi) \tag{10.6}$$

Compare the answer with all points to the answer with every second point:

$$I - T(h) \approx \frac{T(h)-T(2h)}{3},\ I - S(h) \approx \frac{S(h)-S(2h)}{15} \tag{10.7}$$

Equation (10.6) handles an odd number of strips at the same order. Equation (10.7) follows from the error orders: halving h divides the trapezoid error by 4 and the Simpson error by 16. It estimates the error without knowing the truth. Book: Sections 10.4.3 and 10.5.

The Orchard Calculator

Listing 10.1 From `orchard.py`: the composite trapezoidal and Simpson rules for equally spaced data.

```python
def trapezoid(y, h):
    """Composite trapezoidal rule for equally spaced values y."""
    return h * (y[0] / 2 + sum(y[1:-1]) + y[-1] / 2)


def simpson(y, h):
    """Composite Simpson 1/3 rule; needs an even number of intervals."""
    n = len(y) - 1
    if n % 2:
        raise ValueError("Simpson's 1/3 rule needs an even number of intervals")
    odd = sum(y[1:n:2])                          # y1, y3, ..., y(n-1)
    even = sum(y[2:n:2])                         # y2, y4, ..., y(n-2)
    return h / 3 * (y[0] + 4 * odd + 2 * even + y[n])
```

Listing 10.1: the slices `y[1:n:2]` and `y[2:n:2]` pick the odd and even inner values. Simpson refuses an odd number of strips rather than silently returning a wrong area. The data file was prepared from a known river curve, so the true area is known: 7026.8 m^2. Book: Section 10.5.

Running It: the Orchard

```
17 offsets, every 10 m along 160 m of road
rule        every 10 m  every 20 m  error estimate
trapezoid       7023.5      7011.0             4.2
Simpson         7027.7      7039.3            -0.8
area 7027 m2 = 0.70 ha, room for about 70 durian trees at 10 m x 10 m
```

The trapezoid is 3.3 m^2 short and its estimate says "add about 4.2" — right sign, right size. Simpson is 0.9 m^2 over and says "subtract about 0.8". The corrected answer, 7026.9 m^2, is within a tenth of a square metre of the truth — better than the tape deserves. Each offset was rounded to 0.1 m, which alone moves the area by about a square metre. Book: Section 10.5.

Where the Error Comes From

Around the bend, from 100 to 120 m, the offsets are 53.3, 52.3 and 47.2 m.

Trapezoids: 1025.5 m^2. Simpson: 1032.3 m^2. True: 1031.3 m^2.

The bank bulges outward; straight tops cut off almost six square metres, the parabola follows the bulge to within one.

> **KEY INSIGHT.** For Anto the answer is 0.70 hectare, give or take a few square metres. If the land letter says 0.6 ha, ask the surveyor to look carefully; if it says 0.7, the family can relax.

The rule's error and the measurement's error are now of the same size. Pushing the rule further would be pointless; a better tape would not. Book: Section 10.5.

Checking the Order

```
  n   trapezoid error  ratio    Simpson error  ratio
  2        1.519e-02              1.297e-03
  4        3.877e-03     3.9      1.068e-04    12.1
  8        9.747e-04     4.0      7.350e-06    14.5
 16        2.440e-04     4.0      4.723e-07    15.6
 32        6.103e-05     4.0      2.973e-08    15.9
 64        1.526e-05     4.0      1.862e-09    16.0
```

`quadrature.py` and `Quadrature.java` integrate $1/(1+x)$ from 0 to 1, whose value $\ln 2$ we know. Doubling the strips divides the trapezoid error by 4 and the Simpson error by 16: orders 2 and 4, measured. Book: Section 10.6.

Romberg Integration

With $R_{i,0}$ the trapezoidal rule on 2^i strips, Richardson's table applies unchanged:

$$R_{i,j} = \frac{4^j R_{i,j-1} - R_{i-1,j-1}}{4^j - 1}, \quad j = 1, \ldots, i \tag{10.8}$$

```
n =  1  0.750000000000
n =  2  0.708333333333  0.694444444444
n =  4  0.697023809524  0.693253968254  0.693174603175
n =  8  0.694121850372  0.693154530655  0.693147901481  0.693147477645
n = 16  0.693391202208  0.693147652819  0.693147194297  0.693147183072  0.693147181917
R[4][4] = 0.693147181916745, error 1.4e-09, 17 function values
```

Equation (10.8): column 1 is Simpson; column 2 is Boole's rule; further columns keep cancelling. Each new row reuses all old function values and adds only the new midpoints. Seventeen values give an error of 1.4×10^{-9}*, where the trapezoidal rule alone would need thousands. Book: Section 10.7.*

Gauss Quadrature: Choosing Where to Measure

If we may choose the points as well as the weights, two points integrate cubics exactly:

$$\int_{-1}^{1} f(x)\,dx \approx f\left(-\frac{1}{\sqrt{3}}\right) + f\left(\frac{1}{\sqrt{3}}\right) \tag{10.9}$$

n points at the roots of the Legendre polynomial integrate degree $2n-1$ exactly. Any interval maps to $[-1,1]$:

$$\int_{a}^{b} f(x)\,dx = \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{a+b}{2} + \frac{b-a}{2}t\right)dt \approx \frac{b-a}{2}\sum_{i=1}^{n} w_i f\left(\frac{a+b}{2} + \frac{b-a}{2}t_i\right) \tag{10.10}$$

Equation (10.9): twice the degree of the trapezoidal rule from the same number of points. Equation (10.10) rescales the nodes and weights to $[a,b]$. The catch: Gauss needs f at its points, so it works for formulas, not for Anto's offsets. Book: Section 10.8.

Gauss Against the Rest

points	Gauss-Legendre	error	trapezoid error	Simpson error
1	0.666666666667	2.6e-02		
2	0.692307692308	8.4e-04	5.7e-02	
3	0.693121693122	2.5e-05	1.5e-02	1.3e-03
4	0.693146417445	7.6e-07	6.9e-03	
5	0.693147157853	2.3e-08	3.9e-03	1.1e-04

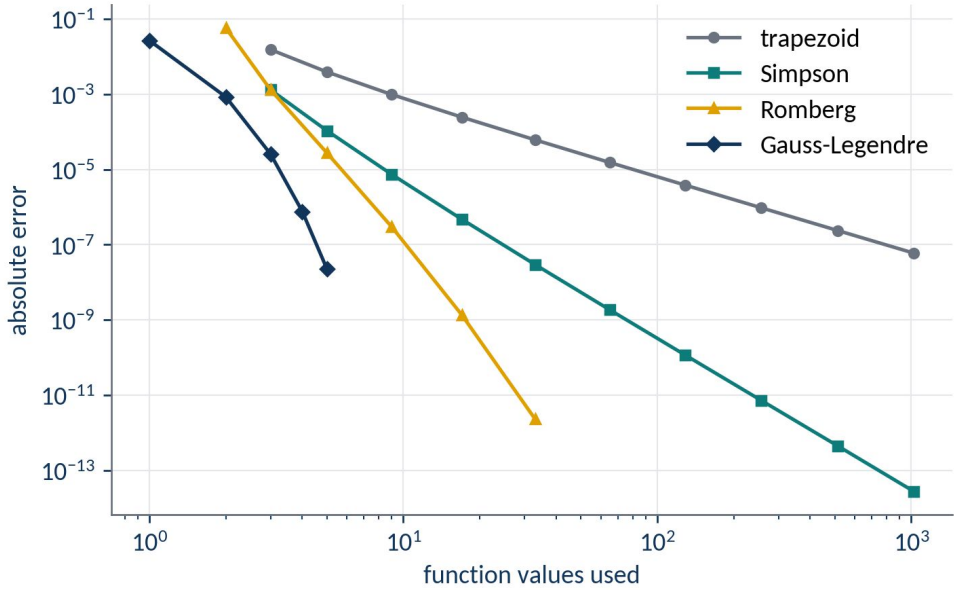


Figure 10.2 Error against the number of function values for $\int_0^1 dx/(1+x) = \ln 2$: trapezoid (order 2), Simpson (order 4), Romberg and Gauss–Legendre.

With five values Gauss–Legendre reaches 2.3×10^{-8}; the trapezoid is at 3.9×10^{-3} and Simpson at 1.1×10^{-4}. Figure 10.2: per function value, Gauss is best, Romberg close behind, and both leave the fixed-order rules far behind. Book: Section 10.8.

Making It Better: Adaptive Simpson

Uniform steps waste effort. Compute Simpson on an interval and on its two halves; if they agree within the tolerance, accept; otherwise split and recurse.

```
tolerance   adaptive result  error    function values
    1e-04   0.666663972068   2.7e-06        37
    1e-06   0.666666647130   2.0e-08       105
    1e-08   0.666666666641   2.6e-11       317
uniform Simpson needs, for an error below 1e-8:
  n = 65536 intervals, error 4.8e-09, 65537 function values
```

The integrand is $\sqrt{x}$ on $[0,1]$, whose infinite slope at 0 spoils the uniform rule's error order. The adaptive rule spends its small steps near 0 and large ones elsewhere: 317 function values where uniform Simpson needs 65,537. Book: Section 10.9.

Checking Against the Library

```
NumPy 2.4.4, SciPy 1.17.1
orchard, trapezoid: 7023.5 m2
orchard, simpson:   7027.7 m2
orchard, romb:      7026.6 m2
quad 1/(1+x):   0.693147180559945 (estimated error 7.7e-15)
fixed_quad n=5: 0.693147157853
quad sqrt(x):   0.666666666666667 (231 function values)
```

For data, `trapezoid`, `simpson` and `romb` take arrays; `romb` needs 2^k+1 equally spaced values, which Anto's 17 offsets happen to be. For functions, `quad` is adaptive Gauss–Kronrod with an error estimate, and `fixed_quad` is our five-point Gauss. `np.trapz` and `integrate.romberg` no longer exist in current releases. Book: Section 10.10.

Numerical Methods in Practice: Profitina

"What was my **average score** this quarter?" An average over time is an integral divided by the length of the period.

With equally spaced weekly scans, the trapezoidal rule gives half weight to the first and last week; with extra scans before a campaign, the trapezoid on the actual dates keeps busy weeks from counting too much.

PROFIT's decaying emission: the total emitted over a period is the integral of the emission rate — trapezoid for tables, Gauss or adaptive rules for smooth formulas.

"Total improvement during a campaign" is the area between the score curve and the starting level. The emission parameters and oracle internals are Profitina's own; the mathematics is this chapter's. Book: Section 10.11.

Common Mistakes

Simpson with an odd number of strips. Use the 3/8 rule for the odd remainder.

Forgetting the halves at the ends. A plain sum times h is the rectangle rule, only first order.

Unequal spacing with an equal-spacing formula. Integrate each strip with its own width.

High-order extrapolation on noisy data. On measured values, stop Romberg after one or two columns.

Expecting Gauss to use your data. It needs values at its own points.

In the field: measure offsets at right angles, at equal spacing, with an even number of strips; then Simpson applies directly and the error estimate is free. Book: Sections 10.12–10.13.

Summary

1. Trapezoid: lines, error $O(h^2)$ (Equation (10.2)). Simpson: parabolas, exact for cubics, $O(h^4)$ (Equation (10.4)); the 3/8 rule for odd strips.

2. Euler–Maclaurin (Equation (10.3)): the trapezoidal error is a series in h^2; Richardson gives Simpson (Equation (10.5)) and Romberg (Equation (10.8)).

3. Halving estimates the error without the truth (Equation (10.7)): 7026.9 m^2 against a true 7026.8 m^2.

4. Gauss–Legendre chooses points and weights, exact to degree $2n-1$ (Equation (10.9), Equation (10.10)); it needs a function, not data.

5. Adaptive Simpson puts small steps only where needed; SciPy provides all of it.

Integration averages errors; that is why a hand-measured orchard came out within a tenth of a square metre. Book: Section 10.14.

Next Time

Chapter 11 — Ordinary Differential Equations: Euler to Runge-Kutta. When the rate is known and the state must be found, step by step.

Euler's method, its error, Heun and the classic fourth-order Runge–Kutta, systems, and stability.

Before then: the first 40 m of the orchard by hand, trapezoid and Simpson, with an error estimate from the difference.

The offsets are 38.6, 40.0, 41.2, 42.2 and 43.1 m; four strips, so Simpson applies. The solution is in the book. Book: Sections 10.15–10.16.

Lecture 11 — Ordinary Differential Equations: Euler to Runge-Kutta

Course: Numerical Methods

Book: Irfan Subakti, *Numerical Methods: From Error Analysis to Machine Learning, with Java and Python*, Chapter 11

Lecturer: Irfan Subakti — yifana@gmail.com

When the rate of change of a quantity depends on the quantity itself, we have a differential equation. Most cannot be solved by formula, but all can be stepped forward in time: from the state now, the state a moment later, and so on. Today: Euler's method, the Runge–Kutta family, a real irrigation morning with switches and overflow, a system that shows what instability looks like, and SciPy's adaptive solver. Book: Chapter 11, Sections 11.1–11.13.

What This Meeting Covers

1. The problem — a drip-irrigation tank: when should the pump stop?

2. The model — a mass balance, an initial value problem, and one exact solution to test against.

3. Euler and Runge–Kutta — one tangent per step, or several slopes per step; orders 1, 2 and 4, measured.

4. The morning simulator — switches, overflow, running dry, and bisection wrapped around a simulation.

5. Systems and stability — a motorbike seat after a speed bump; and `solve_ivp`.

Every program in Java and Python prints the same numbers; the step functions are interchangeable, so one driver compares all methods. Book: Chapter 11.

Will the Tank Last Until Ten?

The dry season at the durian orchard of Chapter 10. Anto has planted **seventy young durian trees** and built a gravity drip system from a **1,000-litre tank** at the top of the slope.

Every morning: pump on at 06:00, drip valve open at 06:30, pump off “when he remembers”.

Some mornings the tank **overflows** — clean well water and the electricity that pumped it, wasted. Other mornings it **runs dry** before ten, and the far trees get too little.

When should he switch the pump off?

He could find out by trial, one morning at a time. But the flow out through the drippers depends on how high the water stands: the rate of change of the level depends on the level itself. That is a differential equation. Book: Section 11.1.

The Idea in Plain Words

The tank has a floor of one square metre. Full, it drips about **10 litres a minute**, which lowers the level by one centimetre a minute.

Pretend the flow stays at 10 L/min for ten minutes: the level falls 10 cm, to 90 cm. But the flow has slowed as the level dropped, so the true fall is a little less.

Recompute the rate at the new level, take another step, and repeat. That is **Euler's method**.

Every method today is a better answer to the same question: which slope should one step use? Book: Section 11.2.

The Model and Its Exact Solution

Mass balance, with floor area A, level h, pump inflow and Torricelli's outflow through the drippers:

$$A\frac{dh}{dt} = Q_{\text{in}}(t) - k\sqrt{h} \tag{11.1}$$

The general **initial value problem**:

$$y'(t) = f(t, y(t)), y(t_0) = y_0 \tag{11.2}$$

With the pump off, separating variables gives

$$h(t) = \left(\sqrt{h_0} - \frac{kt}{2A}\right)^2, 0 \le t \le \frac{2A\sqrt{h_0}}{k} \tag{11.3}$$

Equation (11.1) says: rate of change of the stock equals what comes in minus what goes out. Equation (11.2) is the form every solver accepts. Equation (11.3) is our measuring rod: for Anto's tank it reads $h = (1 - 0.005t)^2$ *with* t *in minutes. Book: Section 11.3.*

Euler's Method

Taylor's theorem with the derivative replaced by f, and the remainder dropped:

$$y_{n+1} = y_n + h f(t_n, y_n) \tag{11.4}$$

Each step errs by $O(h^2)$; there are $O(1/h)$ steps, and the errors can grow along the way:

$$\left| y(t_n) - y_n \right| \leq \frac{hM}{2L}\left(e^{L(t_n - t_0)} - 1\right) \tag{11.5}$$

Equation (11.4): follow the tangent for one step. Equation (11.5): the global error is first order in h; M bounds y'' and L is the Lipschitz constant of f, which says how fast nearby solutions drift apart. Book: Section 11.4.

Running It: Euler on the Draining Tank

```
  t(min)   Euler h(m)   exact h(m)     error
       0     1.000000     1.000000  0.000000
      10     0.900000     0.902500 -0.002500
      20     0.805132     0.810000 -0.004868
      30     0.715403     0.722500 -0.007097
      40     0.630821     0.640000 -0.009179
      50     0.551397     0.562500 -0.011103
```

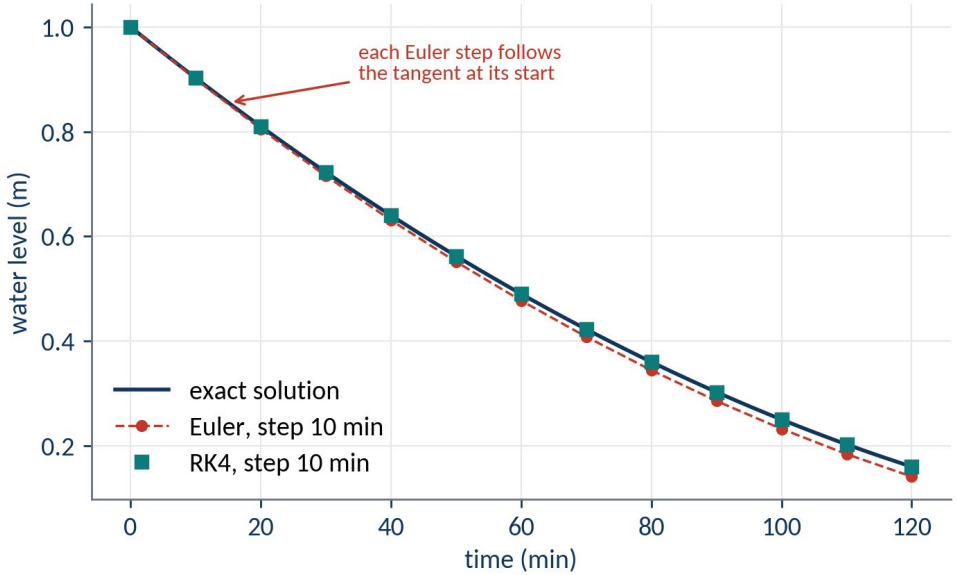


Figure 11.1 The draining tank: the exact solution, Euler's method with a 10-minute step (each step follows the tangent at its start and drains a little too much), and the classical Runge–Kutta method with the same step.

`tank_euler.py` and `TankEuler.java`. *The first step is our hand computation: 0.90 against the exact 0.9025. Figure 11.1: Euler always uses the slope at the start of a step, where the flow is largest, so it drains a little too much every time and the errors accumulate. Book: Section 11.4.*

Runge–Kutta: Sample the Slope Several Times

Two stages, with free parameters chosen by matching Taylor's series to second order:

$$k_1 = f(t_n, y_n),\; k_2 = f(t_n + c\,h,\; y_n + a\,h\,k_1),\; y_{n+1} = y_n + h(b_1 k_1 + b_2 k_2) \tag{11.6}$$

The conditions are $b_1 + b_2 = 1$ and $b_2 c = b_2 a = \frac{1}{2}$. **Heun**: $b_1 = b_2 = \frac{1}{2}$, $c = 1$. **Midpoint**: $b_2 = 1$, $c = \frac{1}{2}$.

Equation (11.6): Heun averages the slope at the start and at the Euler prediction; midpoint uses the slope half-way along. Both are second order: one extra slope evaluation per step, and the error falls with h^2. Book: Section 11.5.1.

The Classical Fourth-Order Runge–Kutta

$$
\begin{aligned}
k_1 &= f(t_n, y_n), & k_2 &= f\left(t_n + \frac{h}{2},\ y_n + \frac{h}{2}k_1\right), \\
k_3 &= f\left(t_n + \frac{h}{2},\ y_n + \frac{h}{2}k_2\right), & k_4 &= f(t_n + h,\ y_n + hk_3), \\
y_{n+1} &= y_n + \frac{h}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)
\end{aligned}
\tag{11.7}
$$

Listing 11.1 From `ode_orders.py`: one RK4 step, interchangeable with the Euler, Heun and midpoint steps.

```
def rk4_step(f, t, y, h):
    k1 = f(t, y)
    k2 = f(t + h / 2, y + h / 2 * k1)
    k3 = f(t + h / 2, y + h / 2 * k2)
    k4 = f(t + h, y + h * k3)
    return y + h * (k1 + 2 * k2 + 2 * k3 + k4) / 6
```

Equation (11.7): four slopes — start, two at the middle, end — weighted 1, 2, 2, 1. When f depends only on t, this is exactly Simpson's rule from Chapter 10. Listing 11.1: a step is a function, so one `solve` driver runs every method. Book: Section 11.5.2.

The Orders, Measured

```
Euler:
  step 10.00   error 1.286e-02
  step  5.00   error 6.335e-03  ratio   2.03
  step  2.50   error 3.144e-03  ratio   2.01
Heun:
  step 10.00   error 3.988e-04
  step  5.00   error 9.666e-05  ratio   4.13
  step  2.50   error 2.380e-05  ratio   4.06
RK4:
  step 10.00   error 1.535e-07
  step  5.00   error 9.345e-09  ratio  16.42
  step  2.50   error 5.762e-10  ratio  16.22
```

Error at 60 minutes for halved steps (the program also prints midpoint and a fourth step size). Ratios 2, 4 and 16: orders 1, 2 and 4. With six 10-minute steps, RK4's error is a sixth of a micrometre for 24 slope evaluations; Euler with 48 evaluations is still wrong by 1.6 mm — ten thousand times worse for twice the work. Book: Section 11.5.3.

The Morning Simulator

At 06:00 the tank holds 25 cm. The pump adds 20 L/min until switched off; the drip valve opens at 06:30.

Above 1 m, water leaves through the overflow; below 1 cm the drippers stop.

RK4 with a one-minute step, and every step **split exactly at a switch**, so no step straddles a sudden change.

> **WARNING.** Runge–Kutta's accuracy assumes a smooth right-hand side within each step. A step that straddles the pump switch mixes two different equations and loses its order. Split the step at every event.

The simulator is Equation (11.1) with a pump term that switches off, a drip term that switches on, and two limits. Book: Section 11.6.

Running It: Two Mornings

```
pump off at 07:30, drip valve open from 06:30
 time   level(cm)  liters
06:00       25.0     250
06:30       85.0     850
07:00      100.0    1000
07:30      100.0    1000
08:00       72.3     723
...
highest level 100.0 cm, spilled 454 liters
drippers stop at 10:31
check with half the step: highest 100.0 cm, spilled 455 liters, dry at 10:30
```

The rows between 08:00 and the summary are left out here. With the pump on until 07:30, the tank is full at about 06:44 and ***454 litres*** *overflow. The exact solution checks the dry time: from full at 07:30,* $1 - 0.005\,s = 0.1$ *gives 180 minutes, 10:30. With the pump off at 06:45, only 5 litres spill, but the drippers stop at 09:46. Book: Section 11.6.*

Planning the Pump: Bisection Around a Simulation

Anto's real question is inverse: *which pump-off time T fills the tank to 98 cm and no higher?* Find the root of $g(T) = \text{top}(T) - 0.98$ — Chapter 2's bisection, with a whole simulation as the function.

```
 step   pump off   highest level(cm)
    1     60.000        100.00
    2     45.000        100.00
    3     37.500         92.44
    4     41.250         96.52
    5     43.125         98.54
...
   12     42.729         98.00
switch the pump off after 42.7 min, at about 06:42: top 98.0 cm, spilled 0 liters, drippers stop at
09:41
```

`pump_plan.py` and `PumpPlan.java`. Twelve halvings bring the interval to 0.015 minutes, far finer than anyone switches a pump. Splitting the step at the switch matters here: otherwise g would move in jumps and bisection could not locate its root. Book: Section 11.7.

The Planned Morning

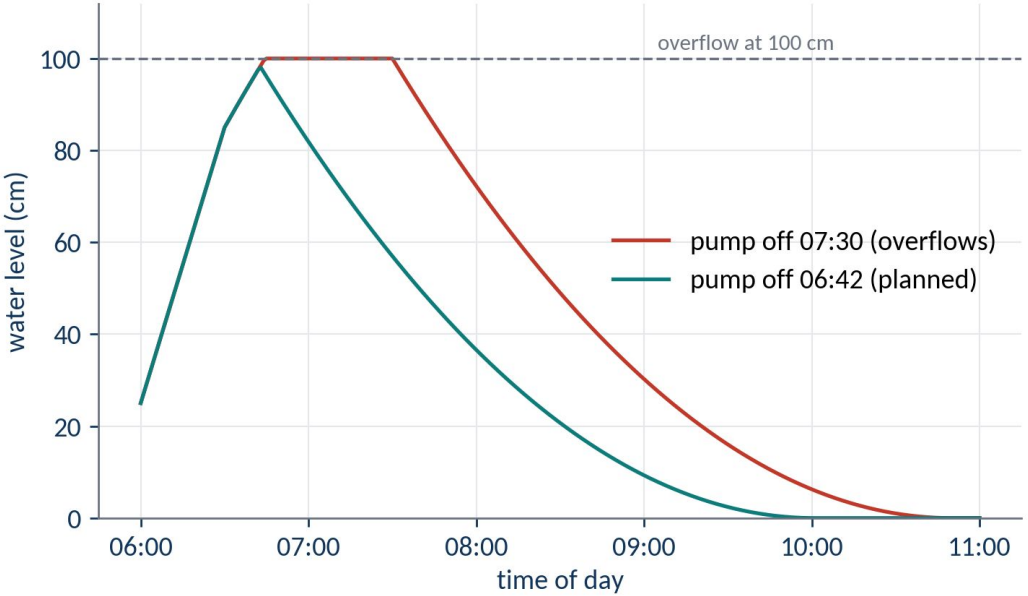


Figure 11.2 Water level on two mornings: with the pump left on until 07:30 the tank overflows for three quarters of an hour; with the pump switched off at 06:42, as planned by bisection, it peaks at 98 cm and never overflows.

Figure 11.2: the plan avoids waste but ends the watering at 09:41. The way out is a pump that delivers less for longer, for instance by partly closing a valve on its outlet; an exercise finds that flow with the same planner. Book: Section 11.7.

Systems: the Motorbike Seat

A seat on a spring and a shock absorber after a speed bump: $m\,x'' + c\,x' + k\,x = 0$. Introduce the velocity v:

$$x' = v\,,\quad v' = -\frac{c\,v + k\,x}{m} \tag{11.8}$$

seat position (mm) at t =	0.5 s	1.0 s	1.5 s	2.0 s
good shock, Euler h=0.01	6.401	0.509	0.003	-0.007
good shock, RK4 h=0.01	4.386	0.344	0.024	0.001
worn shock, Euler h=0.01	42.236	3.606	-48.413	-82.305
worn shock, RK4 h=0.01	25.762	1.608	-10.594	-11.134
worn shock, RK4 h=0.001	25.761	1.607	-10.595	-11.134

Equation (11.8): any higher-order equation becomes a first-order system; Euler and RK4 apply unchanged, with vectors. With the worn shock, Euler's swing grows to 82 mm while RK4, confirmed by a ten times smaller step, shows it dying away. Book: Section 11.8.1.

Why Euler Blew Up: Stability

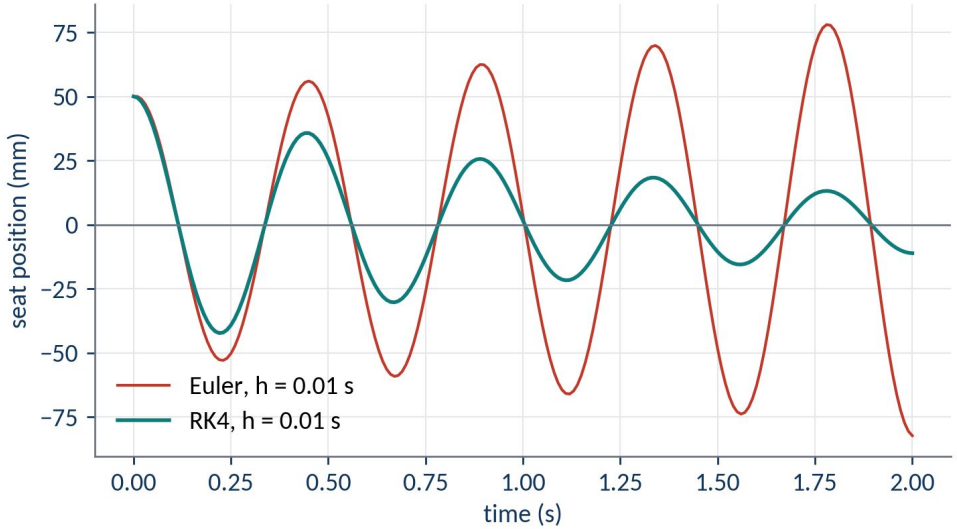


Figure 11.3 The seat with a worn shock absorber after a speed bump: RK4 shows the swing dying away, while Euler's method with the same step makes it grow, an instability of the method, not of the motorbike.

On the test equation $y' = \lambda y$, Euler multiplies by $1 + h\lambda$ each step. It decays only if

$$|1 + h\lambda| \le 1 \tag{11.9}$$

*Figure 11.3: the same step, two methods, opposite behaviour. Equation (11.9) is a disc of radius 1 centred at -1. For the seat it requires $h \le 2\zeta/\omega$: 0.05 s for the good shock, **0.0075 s** for the worn one, so $h = 0.01$ s is outside, and each step multiplies the swing by about 1.0025. Lightly damped oscillations are Euler's worst case. Book: Section 11.8.2.*

Stiff Problems, and Estimating the Error

> **KEY INSIGHT.** An unstable method produces smooth, plausible-looking numbers that are wrong. When a simulation does something the physics cannot — a suspension bouncing more and more, a level going negative — suspect the step before the physics, and halve it.

Halving the step estimates the error of a method of order p:

$$y(t) - y_{h/2} \approx \frac{y_{h/2} - y_h}{2^p - 1} \tag{11.10}$$

*Problems with both very fast and very slow processes are **stiff**: stability forces explicit methods to tiny steps long after the fast process has died out. Implicit methods such as backward Euler solve an equation each step and remain stable. Equation (11.10) is Chapter 10's error estimate for ODEs. Book: Sections 11.8.2 and 11.9.*

Checking Against the Library

```
NumPy 2.4.4, SciPy 1.17.1
tank, RK45 rtol=1e-03: h(60) = 0.490000035, error 3.5e-08, 26 slope evaluations
tank, RK45 rtol=1e-06: h(60) = 0.490000020, error 2.0e-08, 38 slope evaluations
tank, RK45 rtol=1e-09: h(60) = 0.490000000, error 5.7e-11, 68 slope evaluations
worn shock, DOP853, seat position (mm): 25.761  1.607  -10.595  -11.134
```

`solve_ivp`'s default `"RK45"` is the Dormand–Prince pair: a fifth- and a fourth-order formula sharing their slopes, whose difference estimates the error and sets the next step automatically. `DOP853` reproduces our RK4 seat to the last printed digit; `"Radau"` and `"BDF"` are implicit methods for stiff problems. Book: Section 11.9.

Numerical Methods in Practice: Profitina

PROFIT's decaying emission: the rate obeys $E' = -\lambda E$, and the supply obeys $S' = E(t) - B(t)$, emission minus burning. With real, irregular burns it is stepped forward like the tank — events not straddled, projections checked by halving.

The familiar **exponential moving average** $y_{n+1} = y_n + \alpha(x_n - y_n)$ is Euler's method on a first-order lag with $\alpha = h/\tau$.

Seen that way, it is stable and smooth for α between 0 and 1, and would oscillate and blow up above 2 — Equation (11.9) with $\lambda = -1/\tau$.

Smoothed displays such as a weekly score history or a PROFIT↔IDR rate follow raw values with a lag of this kind. How Profitina computes its own figures is its business; the shape of the mathematics is this chapter's. Book: Section 11.10.

Common Mistakes

Trusting Euler because the steps are small. Accuracy is expensive at first order, and oscillations can be unstable. Use RK4 or an adaptive solver.

Straddling events. Split the step at every switch.

Never checking. Compare with an exact case, or halve the step.

Wrong units. Write the balance in words, then symbols, and check the units of every term.

Most practical ODEs are balances — water in a tank, money in an account, stock in a warehouse, charge in a battery, people in a queue. Book: Sections 11.11–11.12.

Summary

1. An initial value problem (Equation (11.2)) gives the rate from time and state; the tank (Equation (11.1)) has an exact solution (Equation (11.3)) to test against.

2. Euler (Equation (11.4)) is first order (Equation (11.5)); two-stage methods (Equation (11.6)) are second order; RK4 (Equation (11.7)) is fourth order.

3. The simulator splits steps at switches; bisection around it finds the pump-off time, 06:42.

4. A second-order equation becomes a system (Equation (11.8)); Euler was unstable on the worn shock (Equation (11.9)), RK4 accurate.

5. Halving estimates the error (Equation (11.10)); `solve_ivp` does it automatically.

Step forward, check by halving, respect events, and watch for instability: that is how every simulation in engineering and science is kept honest. Book: Section 11.13.

Next Time

Chapter 12 — Eigenvalues: the Power Method and Its Uses. Which page matters most? Which state does a system settle into?

The power method, inverse and shifted iteration, and PageRank.

Before then: two Heun steps of 10 minutes by hand for the draining tank, compared with the exact level and with Euler.

The solutions also derive the semi-implicit Euler method, which stays stable on the worn shock with the same step. Book: Sections 11.14–11.15.

Lecture 12 — Eigenvalues: the Power Method and Its Uses

Course: Numerical Methods

Book: Irfan Subakti, *Numerical Methods: From Error Analysis to Machine Learning, with Java and Python*, Chapter 12

Lecturer: Irfan Subakti — yifana@gmail.com

Some questions are not about a particular moment but about what a system does in the long run or by its nature: which share of customers each shop ends up with, at which frequencies a building likes to sway, which pages a random reader lands on most. Each answer is an eigenvalue or an eigenvector. Today we find them with the simplest idea possible — multiply, again and again — and its sharper variants. Book: Chapter 12, Sections 12.1–12.11.

What This Meeting Covers

1. The problem — three laundries in one kampung and customers who switch.

2. Eigenvectors in plain words — directions that a matrix only stretches.

3. The power method — convergence rate, the Rayleigh quotient, inverse and shifted iteration.

4. A shophouse that sways — natural frequencies and resonance.

5. PageRank — the eigenvector of the web; and `numpy.linalg`.

Three programs in Java and Python — `market`, `power`, `pagerank` — all built on one loop: multiply by the matrix, normalize, repeat. Book: Chapter 12.

Three Laundries in One Kampung

In a kampung in **Malang**, Laundry Bersih by the mosque, Laundry Kilat on the main road, and Laundry Wangi, the newest, near the school, share a few hundred households.

Customers are not loyal forever. Bu Endang, who owns Bersih, wonders: will Bersih keep its share, grow, or slowly lose customers to the newcomer?

A survey of 200 households: of Bersih's customers, **80% stay**, 10% go to Kilat, 10% to Wangi. Kilat keeps 75%; Wangi only 60%.

The survey asked which laundry each household used last month and this month. Dividing the counts by each laundry's total gives the fractions that move. Book: Section 12.1.

Eigenvectors in Plain Words

DEFINITION. An **eigenvector** v of a square matrix A is a non-zero vector that A only stretches; the stretch factor λ is its **eigenvalue**.

$$A v = \lambda v \tag{12.1}$$

Example: A with rows $(2,1)$ and $(1,2)$ doubles-and-adds; $(1,1)$ is stretched by 3, $(1,-1)$ by 1. Every other vector turns.

Equation (12.1): eigenvalues describe what a system does when left to itself — which patterns grow, which die out, and how fast. Book: Section 12.2.

The Laundry Market as a Markov Chain

$$P = \begin{pmatrix} 0.80 & 0.15 & 0.20 \\ 0.10 & 0.75 & 0.20 \\ 0.10 & 0.10 & 0.60 \end{pmatrix}, \quad x_{k+1} = P\,x_k \tag{12.2}$$

Column j says where laundry j's customers go; each column adds up to 1. The shares next month are P times the shares now.

Write the start as a combination of eigenvectors, with the stationary shares s for $\lambda_1 = 1$:

$$x_k = P^k x_0 = s + c_2 \lambda_2^k v_2 + c_3 \lambda_3^k v_3 \to s \tag{12.3}$$

*Equation (12.2): a **Markov chain**. Equation (12.3): the other eigenvalues are smaller than 1, so their terms die away and every starting mix ends at s. The distance to s shrinks like the size of λ_2 to the power k. Book: Section 12.3.*

The Market in Code

Listing 12.1 From `market.py`: the shares month by month — the power method on the switching matrix.

```
def main():
    x = [0.40, 0.35, 0.25]                  # shares this month
    print("month   " + "   ".join(f"{n:>7}" for n in NAMES) + "     change")
    print(f"{0:5d}   " + "   ".join(f"{v:7.4f}" for v in x))
    changes = []
    for month in range(1, 16):
        new = mat_vec(P, x)
        change = sum(abs(new[i] - x[i]) for i in range(3))
        changes.append(change)
        x = new
```

Listing 12.1: no eigenvalue theory in sight, just "next month is P times this month". The printing lines and the last ratio are left out here. A stochastic matrix preserves the sum of the shares, so this loop needs no normalization. Book: Section 12.3.

Running It: Where the Customers End Up

```
month   Bersih    Kilat    Wangi     change
    0   0.4000   0.3500   0.2500
    1   0.4225   0.3525   0.2250   5.00e-02
    2   0.4359   0.3516   0.2125   2.68e-02
    3   0.4439   0.3498   0.2063   1.61e-02
    4   0.4489   0.3480   0.2031   9.86e-03
    5   0.4519   0.3465   0.2016   6.10e-03
   ...
   15   0.4571   0.3429   0.2000   6.91e-05
change shrinks by a factor 0.6471 per month
long-run shares: Bersih 45.7%, Kilat 34.3%, Wangi 20.0%
```

Rows 6 to 14 are left out here. Bersih grows from 40% to ***45.7%****, Wangi settles at 20%. The change shrinks by about 0.65 per month — the second eigenvalue, 0.65, measured without computing it. The exact stationary shares are* $(16/35, 12/35, 7/35)$*. Book: Section 12.3.*

The Shares on One Plot

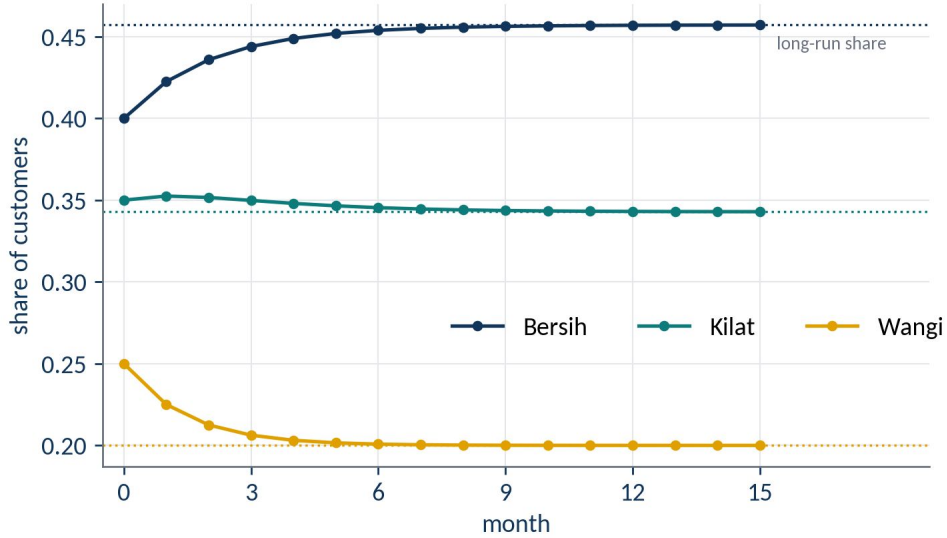


Figure 12.1 Shares of the three laundries month by month from 40%, 35% and 25%; the dotted lines are the stationary shares, the eigenvector of the switching matrix for the eigenvalue 1.

Figure 12.1: within a year the shares are within a few tenths of a percent of the long-run values, whatever the starting mix. For Bu Endang the message is encouraging but not dramatic: if customers keep switching as they did this month, Bersih grows from 40% to about 46%, mostly at

Wangi's expense. The long-run shares depend only on the switching fractions, not on the starting mix; a what-if run with a changed entry, such as Wangi keeping 75% instead of 60%, shows the market after such a change. Book: Section 12.3.

The Power Method in General

For any matrix with one dominant eigenvalue: repeat $x \leftarrow A\,x / \|A\,x\|$. The vector turns toward v_1 at the rate of the size of λ_2/λ_1.

For symmetric A, the **Rayleigh quotient** estimates the eigenvalue with *twice* the rate:

$$r(x) = \frac{x^T A\,x}{x^T x} \tag{12.4}$$

To find the eigenvalue nearest a **shift** σ, apply the power method to the inverse:

$$(A - \sigma I)^{-1} v = \frac{1}{\lambda - \sigma} v \tag{12.5}$$

Equation (12.4): an error of size ϵ in the vector gives an error of size ϵ^2 in $r(x)$. Equation (12.5): the eigenvalue closest to σ becomes the dominant one of the inverse. Factor $A - \sigma I$ once by LU (Chapter 6), then each step is two cheap triangular solves. Book: Section 12.4.

How a Shophouse Sways

Indonesia lies where tectonic plates meet. A three-storey **ruko** — shop below, family above — modelled as three floor masses on elastic columns.

Stiffness matrix K with rows $(2,-1,0)$, $(-1,2,-1)$, $(0,-1,1)$ times k; each floor mass m. Free vibration gives

$$\frac{1}{m}K v=\omega^2 v, f=\frac{\omega}{2\pi}=\frac{\sqrt{\lambda}}{2\pi} \tag{12.6}$$

*Equation (12.6): each eigenvalue is a squared natural angular frequency, each eigenvector a **mode shape** — a pattern in which all floors move in step. The model only shows the idea; any question about the safety of a real building belongs to a licensed structural engineer. Book: Sections 12.5.1–12.5.2.*

Running It: Three Modes

```
highest lambda   6493.959  f 12.83 Hz  iterations  18  shape  1.802  -2.247   1.000
lowest  lambda    396.125  f  3.17 Hz  iterations   7  shape  0.445   0.802   1.000
middle  lambda   3109.916  f  8.88 Hz  iterations   6  shape -1.247  -0.555   1.000
```

`power.py` and `Power.java`. *The power method needs 18 steps: the Rayleigh quotient improves by* $(3110/6494)^2 = 0.23$ *per step. Inverse iteration needs 7: ratio* $(396/3110)^2 = 0.016$. *Shifted inverse iteration with* $\sigma = 3000$ *needs 6: ratio* $(110/2604)^2 = 0.0018$. *A good shift turns a slow method into a fast one. Book: Section 12.5.2.*

The Mode Shapes, and Why They Matter

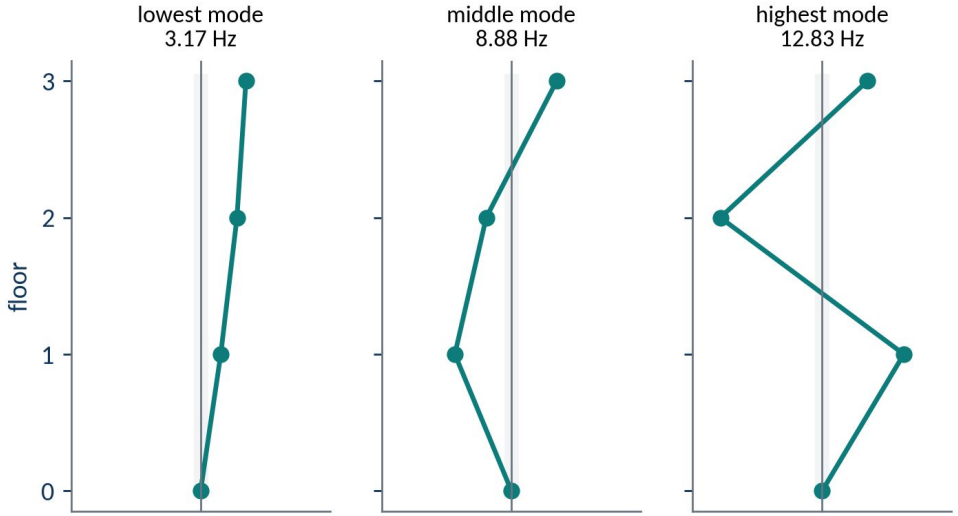


Figure 12.2 The three mode shapes of the shophouse model: sideways displacement of each floor (ground at 0), scaled so that the roof moves one unit, with the natural frequency of each mode.

KEY INSIGHT. Shaken near its natural frequency, a building swings far more than the ground does. With 5% damping, shaking at 1 Hz amplifies about 1.1 times; at 3 Hz about 7 times; at 3.17 Hz, 10 times. The same earthquake can be a gentle rocking for one building and a dangerous swing for another, depending only on an eigenvalue.

Figure 12.2: in the lowest mode all floors move the same way; in the higher modes they move against each other. A heavier roof — a water tank, a drying area — turns the problem into a generalized one, $K v = \omega^2 M v$, and lowers the fundamental frequency from 3.17 to 2.80 Hz. Book: Sections 12.5.3–12.5.4.

PageRank: the Eigenvector of the Web

A reader browses at random: with probability d she follows a random link, otherwise she jumps to any page. A page with no links out (**dangling**) sends her anywhere.

$$r_i = \frac{1-d}{n} + d\sum_{j \to i} \frac{r_j}{L_j} + d\sum_{j\,\text{dangling}} \frac{r_j}{n} \tag{12.7}$$

Equation (12.7): L_j is the number of links out of page j. The ranks are the stationary distribution of this random reader — the eigenvector for eigenvalue 1 of the Google matrix. With $d = 0.85$, as in the original paper, the power method converges at least at the rate d. Book: Section 12.6.

Running It: a Village-Tourism Web

```
converged after 25 iterations, ranks sum to 1.000000000000
1. Village portal       0.2168   (linked from 4)
2. Durian orchard tour  0.1969   (linked from 5)
3. Homestay Melati      0.1757   (linked from 5)
4. Warung Bu Tini       0.1585   (linked from 3)
5. Waterfall guide      0.0922   (linked from 3)
6. Ojek wisata          0.0659   (linked from 1)
```

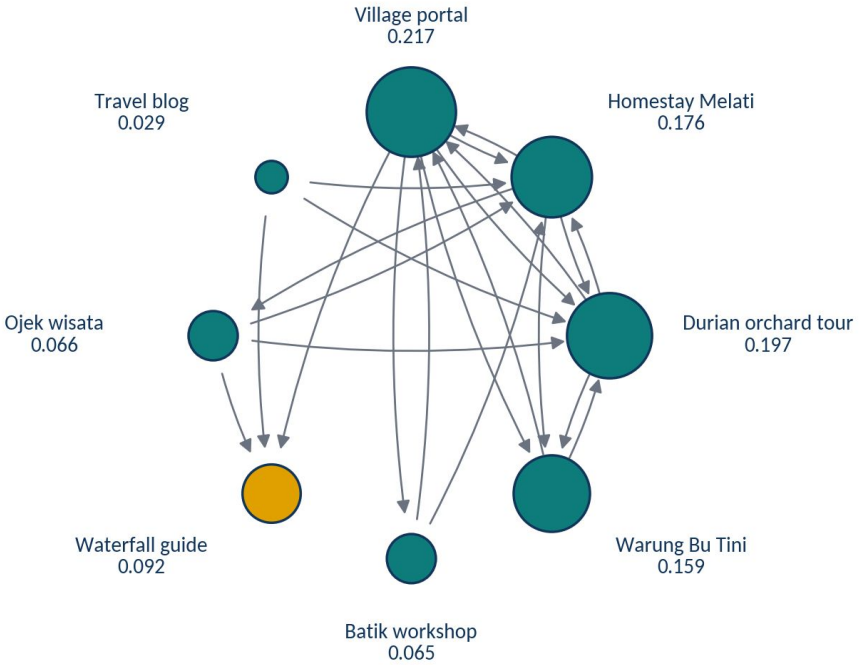


Figure 12.3 The village-tourism network: arrows are links, circle areas are proportional to PageRank, and the waterfall guide (amber) is a dangling page with no links out.

The first six of eight pages. The portal ranks first although two pages have more incoming links: it is linked from the important pages, which pass on a large share of their rank. Figure 12.3: the dangling waterfall guide keeps its rank because the dangling rule spreads it over all pages. Book: Section 12.6.

Why 0.85?

A small d gives the random jump much weight: with $d = 0.5$ the eight ranks lie between 0.07 and 0.17 and say little about the links.

A d close to 1 lets the links speak almost alone, but the second eigenvalue of the Google matrix approaches 1: the power method slows, and the ranks become sensitive to small changes in the links.

Personalized PageRank always jumps to one chosen page: starting from the orchard tour, which pages will a visitor end up reading?

Modern search engines and AI answer engines combine many signals, of which link structure is one, but the idea that importance is inherited through links and computed as an eigenvector remains one of the central ideas of the web. Book: Section 12.6.

Checking Against the Library

```
NumPy 2.4.4, SciPy 1.17.1
market eigenvalues: [1.   0.65 0.5 ]
stationary shares:  [0.457143 0.342857 0.2     ]
building eigenvalues: [ 396.125 3109.916 6493.959]
frequencies (Hz):     [ 3.17  8.88 12.83]
PageRank: [0.2168 0.1757 0.1969 0.1585 0.0654 0.0922 0.0659 0.0285]
second eigenvalue of G: 0.3847
```

`eig` and `eigvalsh` compute all eigenvalues of a small dense matrix at once with the QR algorithm of LAPACK. The market's second eigenvalue is 0.65, exactly the shrink factor we measured; the building and PageRank agree to every printed digit. The second eigenvalue of this Google matrix, 0.3847, is well below d, so 25 iterations were plenty. For huge sparse matrices, `scipy.sparse.linalg.eigs` finds a few eigenvalues. Book: Section 12.7.

Numerical Methods in Practice: Profitina

Profitina measures how ready a brand's site is to be cited by AI answer engines; one family of signals concerns **technical standing** — how a site is built and connected.

When its scanner crawls a site, it builds the link structure of its pages. "Is the product page the owner cares about buried three clicks deep with few links to it?" is a question about the dominant eigenvector of that graph.

In the real-time community, readers move between channels as customers moved between laundries; the long-run attention per channel is a stationary distribution.

The exact signals and weights are Profitina's own; the eigenvector idea is this chapter's, and so is the insight that the second eigenvalue sets how fast a new channel reaches its long-run share. The numbers advise; the people who run the brand decide. Book: Section 12.8.

Common Mistakes

Mixing row and column conventions. Ours has columns adding to 1; check the sums.

Forgetting to normalize. $A^k x$ overflows or underflows.

Stopping on a small change when eigenvalues are close. With a ratio of 0.999 the vector barely moves long before it is accurate.

Expecting the power method to find a tied or complex eigenvalue. Use a shift, or the libraries' QR algorithm.

Inverting the matrix in inverse iteration. Factor $A - \sigma I$ once and solve.

For your own shop, build the switching table from real counts and say how rough the fractions are when there are only a few dozen customers per shop. Book: Sections 12.9–12.10.

Summary

1. An eigenvector is a direction a matrix only stretches (Equation (12.1)); eigenvalues describe what a system does left to itself.

2. Switching customers form a Markov chain (Equation (12.2)); the stationary shares are the eigenvector for 1, approached like λ_2^k (Equation (12.3)): 45.7%, 34.3%, 20.0%.

3. The power method converges at the rate of λ_2/λ_1; the Rayleigh quotient (Equation (12.4)) doubles it for symmetric matrices.

4. Inverse and shifted iteration (Equation (12.5)) gave the shophouse's frequencies (Equation (12.6)): 3.17, 8.88 and 12.83 Hz.

5. PageRank (Equation (12.7)) is the stationary distribution of a damped random reader; NumPy agrees everywhere.

One loop — multiply, normalize, repeat — answered three very different questions. Book: Section 12.11.

Next Time

Chapter 13 — Optimization: Golden Section, Newton and Gradient Descent. The best price, the cheapest route, the smallest error.

One-dimensional search without derivatives, Newton for minima, and gradient descent in many dimensions — the engine of machine learning.

Before then: the laundry shares after one and two months by hand, and a check that $(16/35, 12/35, 7/35)$ satisfies $Ps = s$.

The solutions also find the second-highest shophouse mode by Hotelling deflation. Book: Sections 12.12–12.13.

Lecture 13 — Optimization: Golden Section, Newton and Gradient Descent

Course: Numerical Methods

Book: Irfan Subakti, *Numerical Methods: From Error Analysis to Machine Learning, with Java and Python*, Chapter 13

Lecturer: Irfan Subakti — yifana@gmail.com

Every small business asks which setting of the knobs it controls gives the best result. Chapters 2 and 3 found where a function is zero; today we find where it is largest or smallest. We start with a search that needs only values, move to Newton's method, and end with gradient descent in several variables — the method that trains every model of the next chapter. Book: Chapter 13, Sections 13.1–13.11.

What This Meeting Covers

1. The problem — the best price for a cup of iced tea.

2. Golden-section search — a bracket that shrinks by 0.618 per evaluation.

3. Newton's method for optima — quadratic convergence, and the runaway beyond an inflection.

4. Several variables — gradient, Hessian, gradient descent, its stability limit and its rate.

5. Running and checking — two prices at once, and SciPy's optimizers.

Maximizing f is minimizing $-f$; we move freely between the two. All programs print identical text in Java and Python. Book: Chapter 13.

Mbak Rina's Price

Mbak Rina sells ***es teh jumbo*** in front of a vocational school in **Sidoarjo**. Each cup costs her about Rp 1,500.

For two years she has charged Rp 3,000. Her neighbour raised his price to Rp 5,000 and still sells plenty. Should she follow?

Too low, and each cup earns almost nothing; too high, and few students buy. In between lies the price with the largest **daily profit**.

Four weeks of trial prices, fitted as in Chapter 8, give an exponential demand curve.

The same shape — more is better up to a point, then worse — appears in a laundry's price per kilogram, an ojek driver's hours, a farmer's fertilizer. Book: Section 13.1.

The Model

Cups sold per day at price p (thousand rupiah):

$$D(p) = A\,e^{-p/s},\; A = 600,\; s = 2.5 \tag{13.1}$$

Daily profit with cost $c = 1.5$ per cup:

$$\pi(p) = (p - c)\,D(p) = (p - c)\,A\,e^{-p/s} \tag{13.2}$$

Equation (13.1): each extra Rp 2,500 on the price divides the sales by e. Equation (13.2): margin times cups. At Rp 3,000 she earns $1.5 \times 600\,e^{-1.2} = 271$ thousand rupiah a day. Calculus gives the answer $p^{*} = c + s = 4$, which lets us check every method. Book: Section 13.1.

The Idea: Squeeze the Peak

A function with a single peak on $[a, b]$ is **unimodal**. Compare two probes $x_1 < x_2$: if the profit at x_1 is higher, the peak is not beyond x_2; otherwise it is not before x_1.

Place the probes symmetrically:

$$x_1 = b - r\,L,\; x_2 = a + r\,L,\; \frac{1}{2} < r < 1 \tag{13.3}$$

Choose r so that one old probe is reused as a new one:

$$r^2 = 1 - r \;\Leftrightarrow\; r^2 + r - 1 = 0 \;\Leftrightarrow\; r = \frac{\sqrt{5}-1}{2} \approx 0.6180340 \tag{13.4}$$

Equation (13.3): $L = b - a$. *Equation (13.4): the golden ratio. Every step costs* ***one*** *new evaluation and keeps 61.8% of the interval — the optimization cousin of bisection. Book: Sections 13.2 and 13.3.1.*

Golden-Section Search

Pseudocode 13.1 Golden-section search for the maximum of a unimodal function on $[a, b]$.

```
GOLDEN-MAX(f, a, b, tol)
 1  r = (sqrt(5) - 1) / 2
 2  x1 = b - r(b - a);  x2 = a + r(b - a)
 3  f1 = f(x1);  f2 = f(x2)
 4  while b - a > tol
 5      if f1 > f2                    // peak is not in (x2, b]
 6          b = x2;  x2 = x1;  f2 = f1
 7          x1 = b - r(b - a);  f1 = f(x1)
 8      else                          // peak is not in [a, x1)
 9          a = x1;  x1 = x2;  f1 = f2
10          x2 = a + r(b - a);  f2 = f(x2)
11  return (a + b) / 2
```

$$L_n = r^n L_0, n \geq \frac{\ln(L_0/\varepsilon)}{\ln(1/r)} \approx 2.078 \ln \frac{L_0}{\varepsilon} \tag{13.5}$$

Pseudocode 13.1: lines 6 and 9 reuse a probe, so each step evaluates f once. Equation (13.5): as with bisection, the number of steps is known in advance. Book: Section 13.3.1.

Running It: the Tea Price

k	a	b	b - a	x1	x2	pi(x1)	pi(x2)
1	2.00000	6.94427	4.94427	3.88854	5.05573	302.5347	282.3644
2	2.00000	5.05573	3.05573	3.16718	3.88854	281.7984	302.5347
3	3.16718	5.05573	1.88854	3.88854	4.33437	302.5347	300.3659
4	3.16718	4.33437	1.16718	3.61301	3.88854	298.8193	302.5347
5	3.61301	4.33437	0.72136	3.88854	4.05883	302.5347	302.7622
6	3.88854	4.33437	0.44582	4.05883	4.16408	302.7622	302.2204

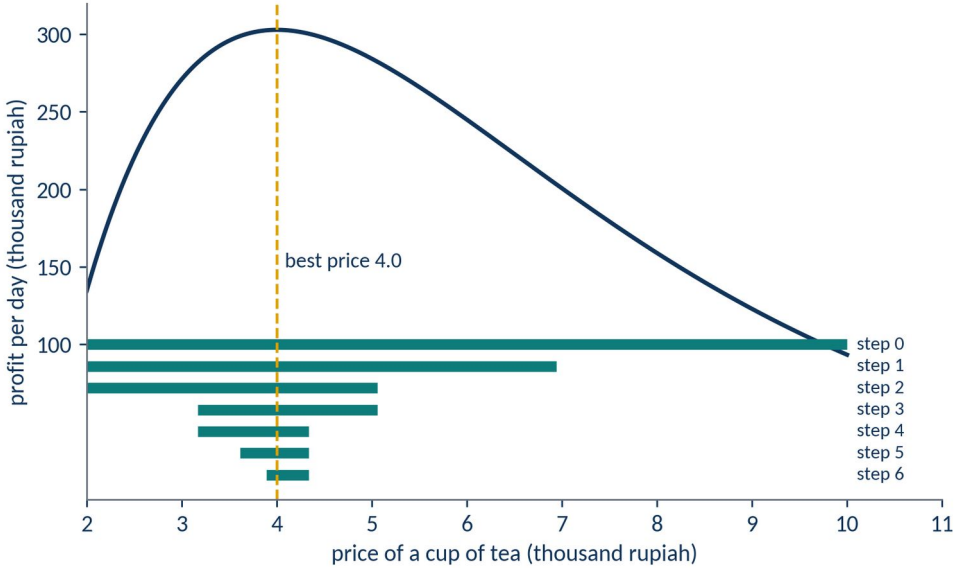


Figure 13.1 Mbak Rina's daily profit and the golden-section intervals of steps 0 to 6: each bar is 0.618 times as long as the one above it, and all of them contain the best price.

The first rows of `golden.py` *and* `Golden.java`*, starting from* $[2, 9]$*. Watch the probe 3.88854 reappear row after row: that is the reuse. Figure 13.1: the bracket always contains Rp 4,000. Book: Section 13.5.*

How Accurately Can a Peak Be Found?

$$\delta_{\min} \approx \sqrt{\epsilon_{\text{mach}}} \sqrt{\frac{2\left|f\left(x^{*}\right)\right|}{\left|f''\left(x^{*}\right)\right|}} \tag{13.6}$$

KEY INSIGHT. Near a peak the function is flat: moving by δ changes it only by about $\frac{1}{2} f'' \delta^2$. Once that change is below rounding, the comparisons stop meaning anything. A search on values can locate an optimum only to about the square root of machine precision — about eight digits.

Equation (13.6): the same flatness is good news in practice — a round price near the optimum loses almost nothing. For Mbak Rina, Rp 4,000 is both optimal and easy to give change for. Book: Section 13.3.1.

Newton's Method for an Optimum

At the optimum $f'=0$: apply Chapter 3's Newton to f'. Equivalently, fit the parabola of Taylor's second-order expansion and jump to its top:

$$x_{k+1}=x_k-\frac{f'(x_k)}{f''(x_k)} \tag{13.7}$$

For the tea profit, with margin $m=p-c$:

$$\pi'(p)=A e^{-p/s}\left(1-\frac{m}{s}\right),\pi''(p)=A e^{-p/s}\left(\frac{m}{s^2}-\frac{2}{s}\right) \tag{13.8}$$

$$e_{k+1}=-\frac{e_k^2}{s-e_k} \tag{13.9}$$

Equation (13.7) uses the curvature to size the step. Equation (13.8) and Equation (13.9): for this profit the error obeys an exact recursion — quadratic near the optimum, but for errors above s the sign flips and the iteration runs away. The inflection point is at $m=2s$, a price of 6.5. Book: Section 13.3.2.

Running It: Plain and Safeguarded Newton

```
plain Newton from p0 = 3.00
   1  p =   3.71428571  error = 2.857e-01
   2  p =   3.97069597  error = 2.930e-02
   3  p =   3.99966049  error = 3.395e-04
   4  p =   3.99999995  error = 4.610e-08
   5  p =   4.00000000  error = 8.882e-16
plain Newton from p0 = 8.00
   1  p =  14.66666667  error = 1.067e+01
   2  p =  17.93197279  error = 1.393e+01
   3  p =  20.97868508  error = 1.698e+01
safeguarded Newton on [2, 9]
   1  bisect  p =   3.75000000  bracket [2.00000, 5.50000]
   2  newton  p =   3.97727273  bracket [3.75000, 5.50000]
...
p = 4.0000000000 after 6 steps
```

From 3.00 each error roughly squares the one before. From 8.00, beyond the inflection, Newton walks away forever, adding about three thousand rupiah per step to a price at which nobody buys tea. The safeguarded version keeps a bracket, bisects once, then accepts every Newton step: six steps. Rows are left out where marked. Book: Section 13.5.

Several Variables: Gradient and Hessian

$$\nabla f(x) = \begin{pmatrix} \partial f/\partial x_1 \\ \vdots \\ \partial f/\partial x_n \end{pmatrix}, H(x) = \left[\frac{\partial^2 f}{\partial x_i \partial x_j} \right]_{i,j=1}^{n} \tag{13.10}$$

$$f(x+h) \approx f(x) + \nabla f(x)^T h + \frac{1}{2} h^T H(x) h \tag{13.11}$$

Mbak Rina adds **gorengan**; students who buy tea often buy a snack too, so each price lowers the other product's sales a little:

$$T = 600\, e^{-p_1/2.5 - 0.10\, p_2}, G = 900\, e^{-p_2 - 0.08\, p_1} \tag{13.12}$$

Equation (13.10): the gradient points uphill; the Hessian describes the curvature in every direction. Equation (13.11): near a point every smooth function looks like a quadratic bowl. Equation (13.12): tea T and snacks G, with costs 1.5 and 0.6 thousand rupiah. Book: Section 13.3.3.

Gradient Descent and Its Stability Limit

$$x_{k+1} = x_k - \alpha \nabla f(x_k) \tag{13.13}$$

On a quadratic bowl the error in each eigen-direction of H is multiplied by $1 - \alpha \lambda_i$ each step:

$$\left|1 - \alpha \lambda_i\right| < 1 \text{ for all } i \Leftrightarrow 0 < \alpha < \frac{2}{\lambda_{\max}} \tag{13.14}$$

$$\rho = \frac{\lambda_{\max} - \lambda_{\min}}{\lambda_{\max} + \lambda_{\min}} = \frac{\kappa - 1}{\kappa + 1}, \kappa = \frac{\lambda_{\max}}{\lambda_{\min}} \tag{13.15}$$

Equation (13.13): walk downhill (for profit, minimize $-\pi$). Equation (13.14) is Chapter 11's stability region in another guise. Equation (13.15): with the best fixed step, the rate is set by the Hessian's ***condition number*** *— a round bowl needs one step, a long narrow valley crawls. Book: Section 13.3.4.*

Choosing the Step Automatically; Newton in n Dimensions

Backtracking: try a step, halve it until the decrease is at least a small fraction of what the gradient promises (Armijo):

$$f(x_k - \alpha \nabla f_k) \le f(x_k) - c\,\alpha \|\nabla f_k\|^2, c = 10^{-4} \tag{13.16}$$

Newton: solve a linear system with the Hessian at every step:

$$H(x_k) s_k = \nabla f(x_k), x_{k+1} = x_k - s_k \tag{13.17}$$

Equation (13.16) never takes an unstable step, whatever the start. Equation (13.17): quadratic convergence whatever the condition number, at the cost of second derivatives and a Chapter 4 solve per step, and only where H is positive definite. Book: Sections 13.3.4–13.3.5.

The Methods Side by Side

Table 13.1 The optimizers of this chapter: what they need, what a step costs, and how fast they converge.

Method	Needs	Cost per step	Convergence	Main danger
Golden section	values, a bracket	1 evaluation	linear, rate 0.618	several peaks
Newton (1 variable)	f', f''	2 derivatives	quadratic near x^*	wrong side of an inflection
Gradient descent	∇f	1 gradient	linear, $(\kappa-1)/(\kappa+1)$	step above $2/\lambda_{\max}$
Newton (n variables)	∇f, H	gradient, Hessian, solve	quadratic near x^*	H not positive definite

Table 13.1 mirrors Chapter 3's table for root finders: safe and slow against fast and fragile, with safeguards to combine the two. Book: Section 13.3.

Running It: Two Prices

```
gradient at (3, 1), formula           (-57.073365, -131.739616)
gradient at (3, 1), central difference (-57.073365, -131.739616)
fixed step 0.002: p = (3.769928, 1.435197) after 166 steps
fixed step 0.008: p = (3.769928, 1.435197) after 35 steps
fixed step 0.012: no convergence in 2000 steps, |grad| = 43.533
fixed step 0.020: no convergence in 2000 steps, |grad| = 212.849
backtracking  : p = (3.769928, 1.435197) after 35 steps, 294 trial points
...
best prices: tea Rp 3770, snack Rp 1435
sales: 115.1 cups, 158.5 snacks; profit 393.533 thousand rupiah a day
Hessian eigenvalues 49.3370 and 182.0565: condition number 3.690
step limit 2/lambda_max = 0.01099, best fixed step 2/(lo + hi) = 0.00864
```

The gradient formula is checked against a central difference first (Chapter 9). Steps 0.002 and 0.008 converge; 0.012 and 0.020 are above the limit 0.01099 and never do — exactly Equation (13.14). Newton, left out here, reaches a gradient of 3×10^{-12} *in five steps. With* $\kappa = 3.69$*, Equation (13.15) predicts* $\rho = 0.574$*. Book: Section 13.5.*

The Paths on the Profit Landscape

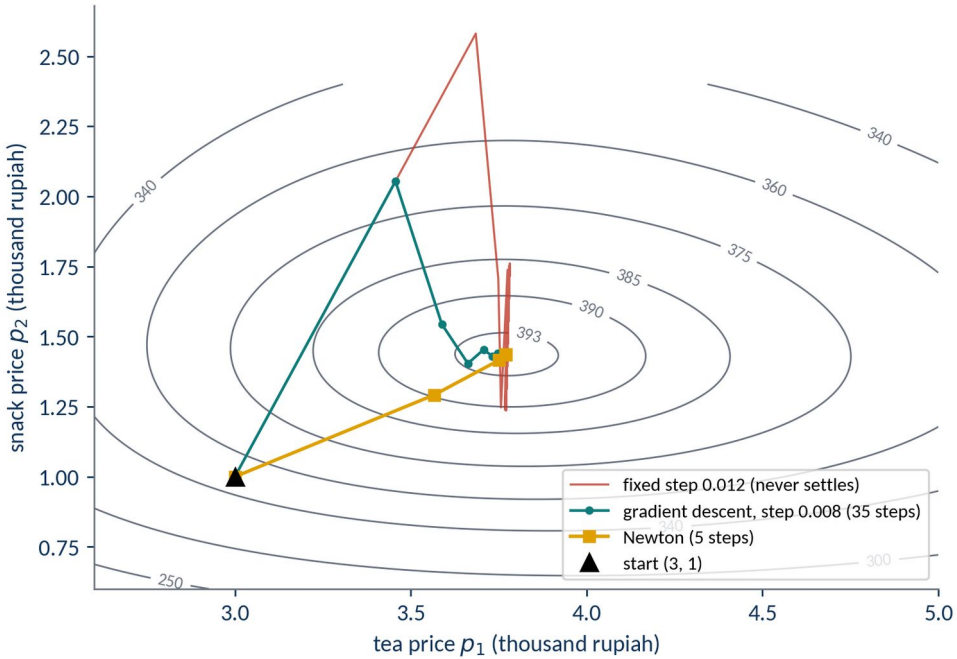


Figure 13.2 Contours of the daily profit (thousand rupiah) over the two prices, with the paths of gradient descent (step 0.008), a fixed step above the stability limit (0.012), and Newton's method, all starting from (3, 1).

Figure 13.2: gradient descent zigzags across the valley; the unstable step bounces from side to side with growing swings; Newton cuts straight to the top. For Mbak Rina: tea at about Rp 3,800 and gorengan at Rp 1,500 — rounding costs well under a thousand rupiah a day because the top is flat. Book: Section 13.5.

Comparing with SciPy

```
SciPy golden : p = 3.99999998, 45 evaluations
SciPy Brent  : p = 4.00000000, 11 evaluations
SciPy BFGS       : p = (3.769928, 1.435197), 9 iterations, 11 evaluations
SciPy Newton-CG  : p = (3.769928, 1.435197), 5 iterations, 6 evaluations
SciPy Nelder-Mead: p = (3.769919, 1.435177), 35 iterations, 67 evaluations
```

`check_scipy.py` *imports our own objective functions, so only the optimizer differs. Brent's method combines golden section with parabolic steps; BFGS builds an approximate Hessian from gradients; Newton-CG is Newton with an iterative solve; Nelder–Mead needs no derivatives at all and is the slowest and least accurate. All libraries minimize: pass* $-\pi$*. Book: Section 13.6.*

Numerical Methods in Practice: Profitina

Profitina re-scans each brand's site at a cadence set by its tier and keeps a weekly score trend. Scanning more often means fresher scores and quicker alerts, but costs server time and loads the customer's website.

As a function of the interval, total cost (staleness plus load) is Mbak Rina's profit upside down: a one-variable minimum, where golden-section search is the natural tool.

Tuning the parameters of PROFIT's decaying emission curve so that supply and demand stay in range is, in the end, minimizing a loss over a few continuous parameters.

The flatness of the minimum is good news: a round interval near it — daily, weekly — loses almost nothing. Choosing among a small set of candidates is optimization over a discrete set: evaluate and compare. Book: Section 13.9.

Common Mistakes

Maximizing when the code minimizes. Pass $-\pi$ and negate the reported value back.

Golden section on a function with several peaks. It finds *a* peak; plot first.

Plain Newton from a poor start. Beyond an inflection it runs away; keep a bracket.

A fixed step above $2/\lambda_{\max}$. Use backtracking.

An untested gradient. Check it against a central difference before optimizing.

For your own stall: change the price for a few days at a time, count what sells, fit the demand curve, and move toward the optimum in small steps — the model advises, the seller decides. Book: Sections 13.8 and 13.10.

Summary

1. Golden section shrinks a bracket by $r = 0.618$ per evaluation (Equation (13.4)); steps known in advance (Equation (13.5)); accuracy limited by flatness (Equation (13.6)).

2. Newton for optima (Equation (13.7)) converges quadratically (Equation (13.9)); a bracket makes it safe.

3. Gradient descent (Equation (13.13)) needs $\alpha < 2/\lambda_{\max}$ (Equation (13.14)); its rate is set by κ (Equation (13.15)); Armijo backtracking chooses safe steps (Equation (13.16)).

4. Newton in n variables (Equation (13.17)) converges quadratically whatever κ.

5. Best tea price Rp 4,000 (302.8 thousand a day); with gorengan Rp 3,770 and Rp 1,435 (393.5 thousand); SciPy agrees.

Gradient descent with a sensible step is the engine of the next chapter. Book: Section 13.11.

Next Time

Chapter 14 — Numerical Methods for Machine Learning. Training a model is least squares, gradients and optimization at scale.

Linear and logistic regression trained by gradient descent, feature scaling, and honest evaluation.

Before then: two more golden-section steps by hand, checked against rows 2 and 3 of the program.

The solutions are in the book in both languages. Book: Sections 13.12–13.13.

Lecture 14 — Numerical Methods for Machine Learning

Course: Numerical Methods

Book: Irfan Subakti, *Numerical Methods: From Error Analysis to Machine Learning, with Java and Python*, Chapter 14

Lecturer: Irfan Subakti — yifana@gmail.com

Machine learning sounds like a new subject. Underneath, a model is a formula with weights, and learning is choosing the weights that make a loss small: least squares from Chapters 6 and 8, gradients and Newton from Chapter 13, the power method from Chapter 12, and the rounding of Chapter 1. Today we train two models on an ojek driver's log and watch every numerical issue the course has met reappear. Book: Chapter 14, Sections 14.1–14.11.

What This Meeting Covers

1. **The problem** — how long will a trip take, and will the customer cancel?
2. **Linear regression** — the loss, its gradient, normal equations against QR.
3. **Training by gradient descent** — the learning rate from the power method, standardization, SGD, momentum and Adam.
4. **Logistic regression** — probabilities, log-loss, Newton's method (IRLS), and stable formulas.
5. **Checking** — NumPy and SciPy, float32 against float64, honest evaluation.

One module per language holds the building blocks; three programs use it; every Java program prints exactly what its Python twin prints. Book: Chapter 14.

Mas Dimas's Two Questions

Mas Dimas drives an ojek in **Yogyakarta**. Two months of trips, **400 rows**: length, rush hour, rain, pickup distance, minutes taken, and whether the customer cancelled.

How long will this trip take? The app's estimate is often wrong in the rain and at rush hour.

Will this customer cancel? A cancellation after three kilometres of pickup costs fuel and a quarter of an hour for nothing.

Both questions are about money, and both are answered by fitting weights to data. The first is a number (regression), the second a probability (classification). Book: Section 14.1.

A Model and a Loss

A **model** turns the **features** of a trip into a prediction — here a weighted sum:

$$\hat{y} = w_0 + w_1\,\mathrm{km} + w_2\,\mathrm{rush} + w_3\,\mathrm{rain} = x^T w \tag{14.1}$$

Learning is choosing w to make the mean squared error small:

$$L(w) = \frac{1}{2N}\sum_{i=1}^{N}\left(x_i^T w - y_i\right)^2 = \frac{1}{2N}\lVert X w - y\rVert^2 \tag{14.2}$$

Equation (14.1): rush and rain are 0 or 1. Equation (14.2) is Chapter 8's least squares with a factor $\frac{1}{2N}$ that makes gradients tidy and independent of the data size. Book: Section 14.2.

Gradient, Hessian, and the Normal Equations

$$\nabla L(w) = \frac{1}{N} X^T (X w - y), \nabla^2 L = \frac{1}{N} X^T X \tag{14.3}$$

$$X^T X w = X^T y \tag{14.4}$$

$$\kappa(X^T X) = \kappa(X)^2 \tag{14.5}$$

$$\frac{\| \hat{w} - w \|}{\| w \|} \lesssim \kappa(X)^2 \epsilon_{\text{mach}} \text{(normal equations)}, \lesssim \kappa(X) \epsilon_{\text{mach}} \text{(QR)} \tag{14.6}$$

Equation (14.3): the Hessian is constant, so the loss is a quadratic bowl. Equation (14.4): gradient zero. Equation (14.5) and Equation (14.6) are Chapter 6's warning: the normal equations lose twice as many digits as QR. Book: Sections 14.3.1–14.3.2.

Running It: the Trip-Time Model

```
400 trips; model minutes = w0 + w1 km + w2 rush + w3 rain
normal equations:   4.00351   2.59547   5.68935   5.52595
Householder QR  :   4.00351   2.59547   5.68935   5.52595
RMSE 2.5693 minutes, R^2 0.9486

polynomials in distance: RMSE (minutes) by method
degree  normal equations   Householder QR
5 km           4.524659         4.524659
9 km           4.503983         4.503983
11 km          4.502751         4.502704
12 km     Cholesky fails         4.494450
12 std         4.494450         4.494450
```

*4.0 minutes of fixed time, **2.6 per kilometre**, 5.7 extra at rush hour, 5.5 in rain; a typical prediction is off by two and a half minutes. At degree 11 the normal equations return an RMSE larger than the true minimum — they have not found the least-squares solution at all; at degree 12 Cholesky fails. With the distance standardized, both methods succeed. Book: Section 14.5.*

Standardize the Features

$$u_{ij} = \frac{x_{ij} - \mu_j}{\sigma_j} \tag{14.7}$$

Every feature gets mean 0 and spread 1. Powers of numbers between about -1.6 and 2.1 are far less alike than powers of numbers between 0.9 and 15.

WARNING. Apply the *training* means and spreads to new data. Recomputing them on the new data silently changes the model.

Equation (14.7): for this data, $\kappa(X^T X)$ falls from 546 to 1.21. The effect on training is dramatic: the safe learning rate rises from about 0.026 to about 1.8, and the bowl becomes almost round, so every direction converges at nearly the same speed. Book: Section 14.3.2.

Training by Gradient Descent

Gradient descent on Equation (14.2) is Chapter 13's method on a quadratic bowl: stable only for a learning rate below $2/\lambda_{\max}$ of $X^T X/N$.

$\lambda_{\max}$ comes from a few steps of the **power method** (Chapter 12) — no guessing.

Mini-batch SGD estimates the gradient from B random rows:

$$w \leftarrow w - \alpha \frac{1}{B} \sum_{i \in B} \left(x_i^T w - y_i \right) x_i \tag{14.8}$$

Equation (14.8): each step is cheap and noisy; with millions of rows, cheap steps win. With a fixed learning rate the noise never dies: SGD ends in a cloud around the minimum, a noise floor. Book: Section 14.3.3.

Momentum and Adam

$$m \leftarrow \beta m + g,\; w \leftarrow w - \alpha m,\; \beta = 0.9 \tag{14.9}$$

$$m \leftarrow \beta_1 m + (1-\beta_1) g,\; v \leftarrow \beta_2 v + (1-\beta_2) g^2,\; w \leftarrow w - \alpha \frac{m/(1-\beta_1^t)}{\sqrt{v/(1-\beta_2^t)} + 10^{-8}} \tag{14.10}$$

Equation (14.9): the step remembers its direction, which damps the zigzag across a narrow valley. Equation (14.10): Adam scales each weight's step by a running estimate of its gradient size — a per-coordinate cure for bad scaling. Both are the default tools of deep learning. Book: Section 14.3.3.

Running It: Training

```
lambda_max of X^T X / N: raw 77.409, standardized 1.092
method              inputs  lr      epochs  excess loss  reached
full-batch GD       raw     0.012      200     2.51e+00        -
full-batch GD       std     0.5        200     0.00e+00       12
mini-batch SGD      std     0.05       200     1.88e-03       92
SGD + momentum      std     0.005      200     2.51e-03       24
Adam                std     0.05       200     4.35e-03       88
```

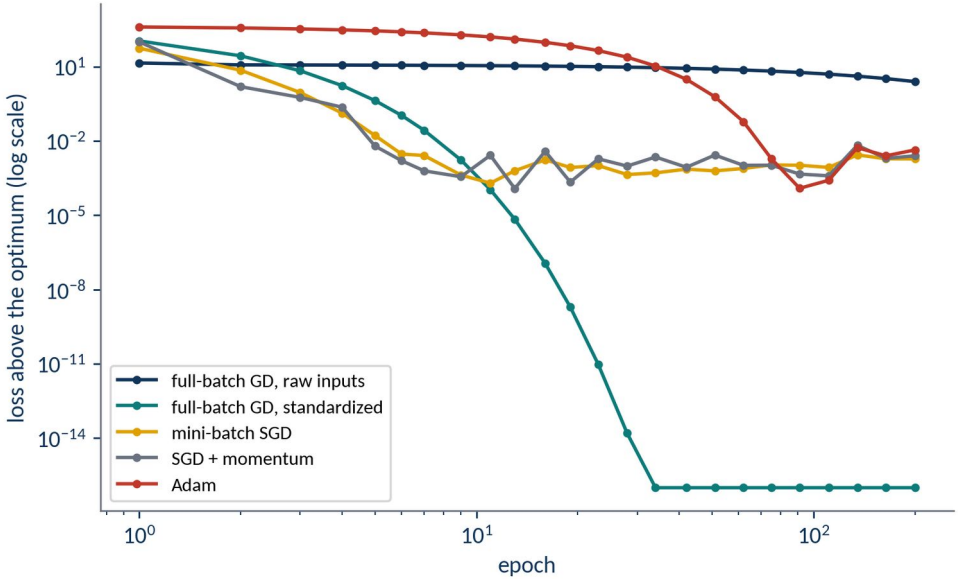


Figure 14.1 Loss above the least-squares optimum during training: full-batch gradient descent on raw and on standardized inputs, and mini-batch SGD, momentum and Adam on standardized inputs (both axes logarithmic).

On raw inputs the safe rate is below $2/77.4$*, and 200 epochs leave the loss 2.51 above the optimum. Standardized, the rate can be 0.5 and the loss reaches the optimum in 12 epochs. Figure 14.1: the stochastic methods drop fast and then level off at their noise floors. Book: Section 14.5.*

Logistic Regression: from a Sum to a Probability

$$p=\sigma(z)=\frac{1}{1+e^{-z}},\sigma'(z)=\sigma(z)(1-\sigma(z)) \tag{14.11}$$

$$L(w)=-\frac{1}{N}\sum_{i=1}^{N}[t_i\ln p_i+(1-t_i)\ln(1-p_i)],p_i=\sigma(x_i^T w) \tag{14.12}$$

$$L(w)=\frac{1}{N}\sum_{i=1}^{N}[\mathrm{softplus}(z_i)-t_i z_i],z_i=x_i^T w \tag{14.13}$$

Equation (14.11): the sigmoid squeezes any weighted sum into $(0,1)$. Equation (14.12): the log-loss punishes confident wrong answers hard; t_i is 1 for a cancellation. Equation (14.13) is the same loss rewritten so that nothing overflows. Book: Section 14.3.4.

Newton's Method: IRLS

$$\nabla L = \frac{1}{N} X^T (p - t), H = \frac{1}{N} X^T W X, W = \mathrm{diag}(p_i(1 - p_i)) \tag{14.14}$$

$$\frac{1}{N} X^T W X s = \frac{1}{N} X^T (p - t), w \leftarrow w - s \tag{14.15}$$

Equation (14.14): the gradient has the same shape as for least squares; the Hessian is positive semidefinite, so the loss is convex and has no false minima. Equation (14.15) is Chapter 13's Newton step — a weighted least-squares problem each iteration, hence iteratively reweighted least squares. Book: Section 14.3.4.

Stable Formulas

$$\operatorname{softplus}(z) = \max(z, 0) + \log 1p\left(e^{-|z|}\right) \tag{14.16}$$

Listing 14.1 From `mlcore.py`: the stable softplus that keeps the log-loss finite.

```
def softplus(z):
    """log(1 + e^z) without overflow: max(z, 0) + log1p(e^-|z|)."""
    return max(z, 0.0) + math.log1p(math.exp(-abs(z)))
```

$$\epsilon_{32} = 2^{-24} \approx 6.0 \times 10^{-8}, \epsilon_{64} = 2^{-53} \approx 1.1 \times 10^{-16} \tag{14.17}$$

Equation (14.16): the exponential is only ever taken of a non-positive number. Listing 14.1: three lines. Equation (14.17): machine learning often computes in float32 for speed — about seven digits, so long sums and subtractions need Chapter 1's care. Book: Sections 14.3.5–14.3.6.

Running It: Will the Customer Cancel?

```
one trip, stable versus naive log-loss
  z =    -5.0, cancelled = 1: naive   5.006715, stable 5.006715
  z =   -40.0, cancelled = 1: naive  40.000000, stable 40.000000
  z =  -800.0, cancelled = 1: naive         inf, stable 800.000000
  z =    40.0, cancelled = 0: naive         inf, stable 40.000000
gradient check, max difference: 2.98e-10
Newton / IRLS from w = 0
  1  loss 0.4596129941  |grad| 5.70e-02
  ...
  6  loss 0.4317353354  |grad| 4.68e-16
weights: -4.80740 1.29936 1.00720 1.11312
odds of cancelling x3.667 per km of pickup, x2.738 in rain, x3.044 in rush hour
```

*The naive formula returns infinity where the true loss is 800 or 40. The gradient check agrees to 3×10^{-10}. Newton needs **six steps**; gradient descent on the standardized pickup needs 128 epochs to come within 10^{-6}. Each kilometre of pickup multiplies the odds of a cancellation by 3.7. Book: Section 14.5.*

The Learned Probabilities

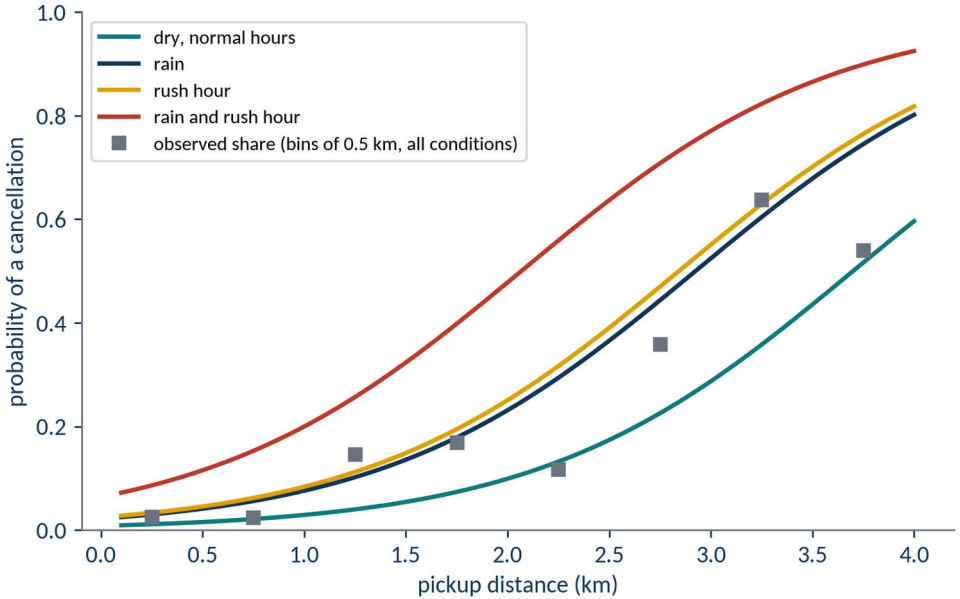


Figure 14.2 Probability that an order is cancelled, as learned by logistic regression, against the pickup distance for four combinations of rain and rush hour; squares show the observed share in half-kilometer bins over all conditions.

KEY INSIGHT. A probability is not yet a decision. With the threshold 0.5 the model is right 79% of the time, catches 57 of 110 cancellations and raises 31 false alarms. Where to put the threshold depends on what a missed cancellation and a refused order each cost Mas Dimas.

Figure 14.2: the curves rise steeply with the pickup distance and sit higher in rain and at rush hour. When only 27.5% of orders are cancelled, "nobody cancels" is already 72.5% accurate — report what is caught and what is falsely flagged, not accuracy alone. Book: Section 14.5.

Comparing with NumPy and SciPy

```
lstsq weights: 4.00351 2.59547 5.68935 5.52595
cond(X^T X): raw 546.1, standardized 1.209
degree  5: cond(X) 5.63e+06, cond(X^T X) 3.17e+13
degree  9: cond(X) 3.15e+12, cond(X^T X) 1.62e+23
degree 11: cond(X) 2.55e+15, cond(X^T X) 9.39e+26
degree 12: cond(X) 7.62e+16, cond(X^T X) 5.73e+28
BFGS logistic weights: -4.80740 1.29936 1.00720 1.11312 loss 0.4317353354
```

`lstsq` *uses the singular value decomposition and agrees. The condition numbers confirm Equation (14.5): at degree 11* $\kappa\left(X^T X\right)$ *is about* 10^{27}*, far beyond* $1/\epsilon_{64}$*, so the normal equations had no digits left. SciPy's BFGS with the library's stable* `log_expit` *finds the same logistic weights. Book: Section 14.6.*

Making It Better

Regularize. Adding $\frac{\lambda}{2}\|w\|^2$ (ridge) shifts every eigenvalue of $X^T X$ up by $N\lambda$: smaller κ, Cholesky cannot fail, weights stop chasing noise.

Hold data back. Fit on part of the log, measure on the rest — Chapter 8's lesson, now called a validation set.

Accumulate carefully. Long sums of losses in float32 lose digits; sum in float64 or pairwise.

The same steps apply to any small business that keeps a log: laundry orders, waiting times in a puskesmas queue, the yield of each row of an orchard. The model advises; the person decides. Book: Sections 14.7–14.8.

Numerical Methods in Practice: Profitina

Profitina's AI features run on **local models only**: Qwen2.5 through Ollama, a 7B model for quality work and an optional small model (qwen3:1.7b) as a fast lane for short classification.

A small model that replies "yes or no" does, at scale, what the cancellation model does: a weighted sum turned into a probability by a sigmoid, or its many-class cousin, the softmax — computed stably by the log-sum-exp form.

The feedback pipeline: user feedback on AI responses is curated into a dataset, the model is trained on it and **measured before a release**.

The Link Safety feature shows that a classifier's threshold is a decision: its rule floor catches 27 of 28 realistic scam texts with no false positives on 30 normal texts in the internal benchmark, and moving the threshold would trade missed scams against false alarms, exactly as Mas Dimas's threshold does. Book: Section 14.9.

Common Mistakes

Solving least squares by inverting $X^T X$. Use QR or a library routine.

Unscaled features. Standardize with Equation (14.7), using the training statistics.

A learning rate copied from a tutorial. Estimate $\lambda_{\max}$ with the power method.

The log-loss as written in the textbook. Use the softplus form.

Reading SGD's noise floor as convergence; trusting accuracy alone.

Every one of these mistakes is a numerical-methods mistake from an earlier chapter wearing new clothes. Book: Section 14.10.

Summary

1. Learning is minimizing a loss: least squares (Equation (14.2)) for numbers, log-loss (Equation (14.12)) for probabilities.

2. The normal equations (Equation (14.4)) square κ (Equation (14.5)); QR does not (Equation (14.6)).

3. Gradient descent's rate is set by κ; the power method gives the safe learning rate; standardization cut κ from 546 to 1.21 and the epochs to 12.

4. SGD (Equation (14.8)), momentum and Adam trade exact steps for cheap ones; IRLS (Equation (14.15)) converged in six steps.

5. Stable formulas (Equation (14.16)) and gradient checks keep training honest. Trip time: about 4.0 + 2.6 per km, +5.7 at rush hour, +5.5 in rain.

The numerical methods are those of Chapters 1, 6, 8, 12 and 13; machine learning is where they meet. Book: Section 14.11.

Next Time

Chapter 15 — Fast and Parallel Numerical Computing. The same answers, sooner.

Where the time goes, memory and caches, vectorization, and many cores working on one problem.

Before then: fit the three-trip line by hand with the distance standardized, and convert it back.

The converted line must be $4.333 + 2.667\,\text{km}$*, the same as without standardization; the solution is in the book. Book: Sections 14.12–14.13.*

Lecture 15 — Fast and Parallel Numerical Computing

Course: Numerical Methods

Book: Irfan Subakti, *Numerical Methods: From Error Analysis to Machine Learning, with Java and Python*, Chapter 15

Lecturer: Irfan Subakti — yifana@gmail.com

A numerical method that is correct but takes an hour is, for most people, a method they will not use. Today we take one computation from Chapter 5, the temperature of a martabak griddle, refine it to every millimetre, and make it more than ten thousand times faster: by understanding where time goes, by vectorizing, by using several cores, and above all by choosing a better algorithm. Along the way, parallel sums turn out to change their answer. Book: Chapter 15, Sections 15.1–15.11.

What This Meeting Covers

1. The problem — an hour for one picture of a griddle.

2. Where time goes — operations against memory traffic: the roofline.

3. Faster code — vectorization, compiled loops, threads, and Amdahl's law.

4. Faster algorithms — Jacobi's sweep count against SOR's, and red-black ordering for parallel SOR.

5. Reproducible sums — naive, chunked, pairwise and Kahan summation.

Every program prints the same numbers in Java and Python, on one thread or several; only the optional timing line differs from machine to machine. Book: Chapter 15.

An Hour for One Picture

Chapter 5's griddle: 40 cm square, edges at **200, 60, 80 and 150 °C**.

A blacksmith who makes griddles for a dozen carts wants the temperature **every millimetre** — a pale spot of one centimetre ruins a martabak. That is $400 \times 400 = 160,000$ unknowns.

Plain Python loops: about 15 million point updates per second. Jacobi needs about **340,000 sweeps** at this resolution: 5.4×10^{10} updates — **an hour** per picture.

He wants twenty pictures before lunch.

Three roads to speed: make each update cheaper, do several updates at once, or need fewer updates. The chapter measures all three on one machine. Book: Section 15.1.

The Idea in Plain Words

A computer spends time on **arithmetic**, on **moving data** from memory, and on **bookkeeping** (an interpreter deciding what each line means).

Remove the bookkeeping: whole-array operations, or a compiled language.

Share the work: several cores on separate strips of the grid — if the method allows it.

Need less work: a method that converges in fewer sweeps.

The last road is usually the longest. A faster language multiplies the speed by a constant; a better algorithm can change how the work grows with the problem. Book: Section 15.2.

A Cost Model: the Roofline

W operations at rate F, Q bytes moved at bandwidth B:

$$T \geq \max\left(\frac{W}{F}, \frac{Q}{B}\right) \tag{15.1}$$

A Jacobi sweep does $4n^2$ operations and streams about $16n^2$ bytes: **0.25 operations per byte**.

A core can compute far faster than memory can deliver at that ratio: Jacobi is **memory-bound**.

Equation (15.1): the ratio W/Q is the arithmetic intensity. Below F/B, faster arithmetic or more cores do not help once the memory channel is full. Drawn on a log-log chart, this bound is the "roofline" that performance engineers use to decide what to optimize. Book: Section 15.3.1.

Vectorization

Listing 15.1 From `griddle_numpy.py`: one NumPy statement updates a whole block of rows.

```
def sweep_rows(u, v, lo, hi):
    """Jacobi update of inner rows lo..hi-1: one line does a whole block."""
    v[lo:hi, 1:-1] = 0.25 * (u[lo - 1:hi - 1, 1:-1] + u[lo + 1:hi + 1, 1:-1]
                             + u[lo:hi, :-2] + u[lo:hi, 2:])
```

Listing 15.1: the shifted slices are the four neighbours of every point at once. The loop over points now runs inside NumPy's compiled code instead of the interpreter. With `lo = 1` and `hi = n + 1` it is the full sweep; with other limits it is one thread's strip. Book: Section 15.3.2.

Parallelism and Amdahl's Law

If a fraction f of the work runs perfectly in parallel on p cores and the rest stays serial:

$$S(p) = \frac{T_1}{T_p} = \frac{1}{(1-f) + f/p} \le \frac{1}{1-f} \tag{15.2}$$

With $f = 0.95$, eight cores give 5.93, and no number of cores gives more than 20.

> **WARNING.** Jacobi divides safely into strips: every thread reads the old grid and writes its own rows of the new one. Gauss–Seidel and SOR update *in place*; split naively over threads they have a **race condition** and give timing-dependent results.

Equation (15.2): the serial part, however small, eventually dominates; synchronization and shared memory bandwidth take a further share. Python threads do pure-Python arithmetic in turns because of the global interpreter lock; NumPy releases it inside its loops. Book: Section 15.3.3.

The Better Algorithm

From Chapter 5's theory, sweeps needed to shrink the error by ε on an $n \times n$ grid:

$$k_J \approx \frac{\ln(1/\varepsilon)}{1-\rho_J} \approx \frac{2(n+1)^2}{\pi^2} \ln\frac{1}{\varepsilon} \tag{15.3}$$

$$k_{\text{SOR}} \approx \frac{n+1}{2\pi} \ln\frac{1}{\varepsilon} \tag{15.4}$$

For $n = 400$: about **340,000** Jacobi sweeps against about **1,200** for optimal SOR.

Equation (15.3) grows like n^2, Equation (15.4) like n. A factor of several hundred — more than any language or core count can give. ***Red-black ordering*** *colours the grid like a chessboard: red points depend only on black ones and vice versa, so each half-sweep of SOR is one vectorized, parallel step. Book: Section 15.3.4.*

Why Parallel Sums Change Their Answer

Floating-point addition is not associative. Adding n numbers left to right can err by

$$|\hat{s} - s| \leq (n-1)u \sum_{i=1}^{n} |x_i| + O(u^2) \tag{15.5}$$

Adding in pairs, then pairs of pairs, only by

$$|\hat{s} - s| \leq \lceil \log_2 n \rceil u \sum_{i=1}^{n} |x_i| + O(u^2) \tag{15.6}$$

Equation (15.5): the bound grows with n. Equation (15.6): it grows only with $\log_2 n$ — for ten million terms, 24 instead of ten million. Splitting a sum over threads changes the order, and therefore the rounding, and therefore the answer. Book: Section 15.3.5.

Kahan's Compensated Summation

Pseudocode 15.1 Kahan's compensated summation: c recovers the rounding error of each addition and feeds it into the next.

```
KAHAN-SUM(x[1..n])
 1  s = 0;  c = 0
 2  for i = 1 to n
 3      y = x[i] - c          // the next term, corrected by what was lost
 4      t = s + y             // big + small: low bits of y are lost here
 5      c = (t - s) - y       // (t - s) is what was really added; minus y
 6      s = t                 //   leaves exactly the part that was lost
 7  return s
```

Pseudocode 15.1 is Chapter 1's laundry-ledger trick, now for parallel work. Its error is about $2u$ times the sum of the absolute values, independent of n. Book: Section 15.3.5.

Running It: Ten Million Times 0.1

```
sum of 10000000 copies of 0.1; exact value 1000000.0
naive, one thread  999999.9998389754  error -1.610e-04
naive, 2 chunks    999999.9999107814  error -8.922e-05
naive, 4 chunks    1000000.0000402561  error +4.026e-05
naive, 8 chunks    1000000.0000223045  error +2.230e-05
naive, 16 chunks   999999.9999899360  error -1.006e-05
pairwise           1000000.0000000000  error +0.000e+00
Kahan              1000000.0000000000  error +0.000e+00
```

Every chunking gives a different wrong answer, sometimes too small, sometimes too large. A program that summed "with as many threads as there are cores" would print one result on a 4-core laptop and another on a 16-core server. Pairwise and Kahan are correct to all printed digits. The rule: fix the chunks by the data, not by the core count. Book: Section 15.5.

Running It: the Fine Griddle

```
grid 400 x 400, 2000 Jacobi sweeps, 1 thread(s)
center 0.000000 C, mean 28.428053 C, last change 3.168e-02
time 0.236 s, 1356.4 million updates/s
grid 400 x 400, 2000 Jacobi sweeps, 2 thread(s)
center 0.000000 C, mean 28.428053 C, last change 3.168e-02
time 0.247 s, 1295.8 million updates/s
grid 400 x 400, red-black SOR, omega 1.984453: 1171 sweeps
center 122.572805 C, mean 122.499983 C
time 0.362 s
```

The Java run on the two-core test machine. Plain Python: 15.6 million updates per second; NumPy: 133 million on one thread; Java: ***1.36 billion****, and no more on two threads — at that rate the loop is limited by memory (Equation (15.1)). Jacobi after 2000 sweeps has not even warmed the centre. Red-black SOR solves the whole plate in 1171 sweeps and about a third of a second. Book: Section 15.5.*

Where the Speed Came From

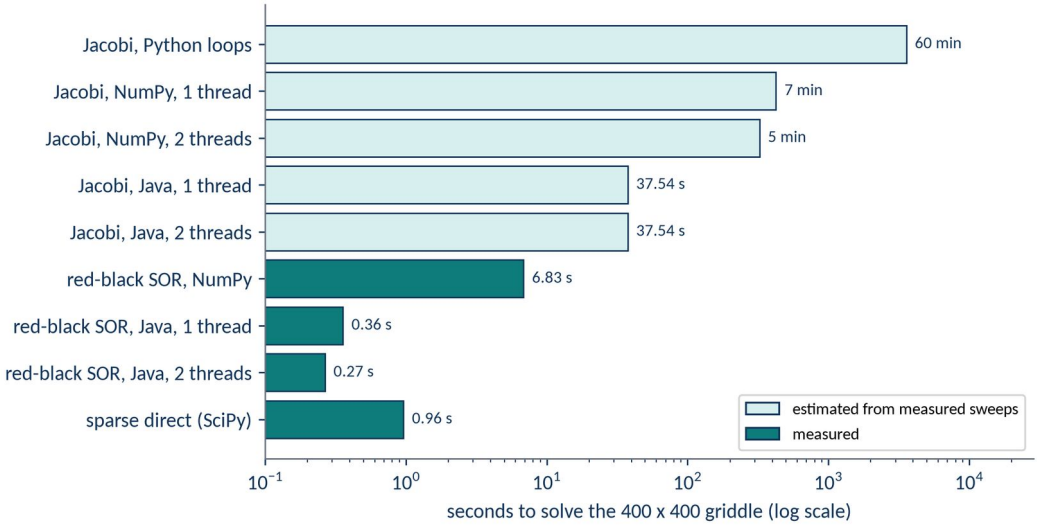


Figure 15.1 Time to solve the 400×400 griddle on the two-core test machine: Jacobi (estimated from the measured time per sweep and the number of sweeps needed), red-black SOR and SciPy's sparse direct solver (measured). The scale is logarithmic.

KEY INSIGHT. From slowest to fastest is a factor of more than ten thousand, and the largest single factor, several hundred, comes from the **algorithm**, not from the language or the cores. Count sweeps before counting nanoseconds.

Figure 15.1: the Jacobi bars multiply the measured time per sweep by 340,000 (an hour of plain Python proves nothing new); SOR and the direct solver are measured runs. Book: Section 15.5.

The Blacksmith's Picture

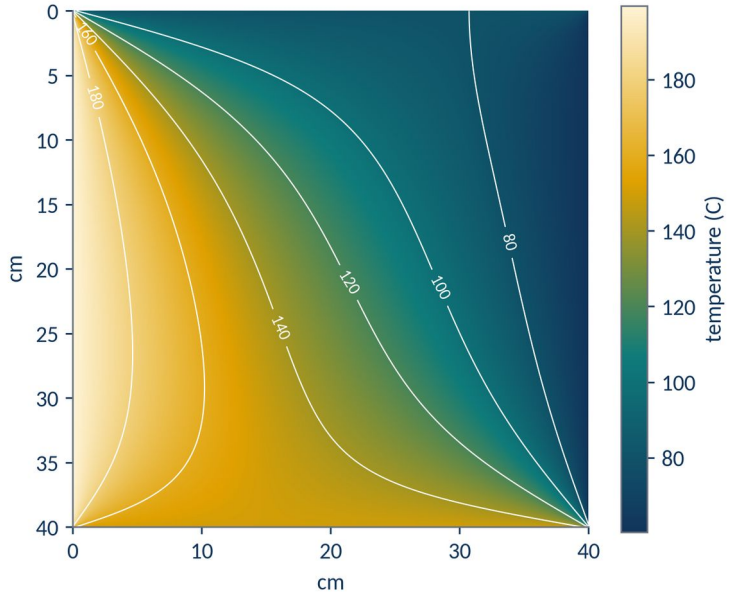


Figure 15.2 Steady temperature of the 40 cm griddle on a 1 mm grid, with the left edge at 200, the right at 60, the top at 80 and the bottom at 150 degrees Celsius; white lines are isotherms.

Figure 15.2: 160,000 temperatures in a third of a second, so twenty burner layouts take seconds, not a day. The mean, 122.5 °C, is exactly the average of the four edge temperatures, as symmetry predicts — a free check of the whole computation. Book: Section 15.5.

Comparing with NumPy and SciPy

```
math.fsum (correctly rounded): 1000000.0000000000
numpy.sum (pairwise)         : 1000000.0000000000
sparse direct solve, 160000 unknowns: center 122.572847 C, mean 122.500000 C, 0.87 s
```

`math.fsum` returns the correctly rounded sum of the exact values; NumPy's `sum` is pairwise and agrees. SciPy's sparse direct solver treats the griddle as the linear system of Chapter 5, built with `kronsum`, and takes 0.87 s; red-black SOR agrees with it at the centre to about 5×10^{-5} *°C. Book: Section 15.6.*

Making It Better

Measure before optimizing. `cProfile` in Python, Java Flight Recorder in the JDK. The first Java run includes compilation; time several runs.

Processes or free-threaded Python for CPU-bound Python code. `ProcessPoolExecutor` runs separate processes; Python 3.14's officially supported free-threaded build removes the interpreter lock, at some cost to single-thread speed.

A better algorithm before a bigger machine. SOR, preconditioned conjugate gradients, multigrid or a sparse direct solver each beat Jacobi by orders of magnitude. Parallelize the good algorithm, not the bad one.

Keep results reproducible. Fix the chunking by the data; seed every random stream from the task, not the thread; keep a single-threaded reference run.

The graphics processor has thousands of simple cores and very high memory bandwidth; stencil sweeps like Jacobi's run there with little change.

The roofline, Amdahl's law and reproducible reductions apply on a GPU unchanged; only the numbers F and B are larger. Book: Section 15.7.

Numerical Methods in Practice: Profitina

Scanning a site is mostly **waiting** for pages to arrive, so many scans can be in progress even on few cores: concurrency limited by input and output, where the question is how many requests a site can fairly be sent at once.

Scoring pages and running the local Qwen2.5 model is **computation**, where Amdahl and the roofline apply: a model's speed is set largely by how fast its weights can be read from memory — hence a very small model as a fast lane.

The release gate — well over a thousand automated tests — is embarrassingly parallel, provided no two tests share state.

Weekly score trends are sums over growing tables and must come out the same whatever the order of the rows: the reproducible summation of this chapter. Book: Section 15.9.

Common Mistakes

Optimizing the wrong thing. A ten times faster Jacobi sweep is still hundreds of times slower than SOR.

Timing once. Repeat, report the best or the median, and name the machine.

Expecting p cores to give p times the speed. Amdahl, synchronization and bandwidth take their share.

Parallelizing an in-place update; letting the thread count decide a sum.

Python threads for pure-Python arithmetic. Use NumPy, processes, or the free-threaded build.

Measure before optimizing: profilers show where the time really goes, and it is rarely where one guesses. The same checklist serves a spreadsheet macro, an orchard simulation or the training of a model. Book: Sections 15.7–15.8 and 15.10.

Summary

1. Time goes to arithmetic, data movement and bookkeeping; the roofline (Equation (15.1)) shows Jacobi is memory-bound.

2. Vectorizing removed the interpreter (15.6 to 133 million updates per second); compiled Java reached 1.36 billion, the memory bound.

3. Amdahl's law (Equation (15.2)) limits parallel speedup; Jacobi splits safely, SOR needs red-black ordering.

4. Jacobi needs about 340,000 sweeps, SOR 1171 (Equation (15.3), Equation (15.4)): the better algorithm beat the faster language and the extra core.

5. Sums depend on order (Equation (15.5)); pairwise (Equation (15.6)) and Kahan (Pseudocode 15.1) make them accurate and reproducible.

Identical digits across Python loops, NumPy and Java came from performing the same operations in the same order. Book: Section 15.11.

Next Time

Chapter 16 — Capstone: Simulating a Real System End to End. One real problem, every tool of the course. Modelling, differential equations, fitting, root finding, optimization and Monte Carlo, checked at every step.

Before then: Amdahl’s law for a loop that is 90% of the time, on 4, 16 and infinitely many cores.

The exercise also asks what fraction must be parallel for 16 cores to give a speedup of 12; the solution is in the book. Book: Sections 15.12–15.13.

Lecture 16 — Capstone: Simulating a Real System End to End

Course: Numerical Methods

Book: Irfan Subakti, *Numerical Methods: From Error Analysis to Machine Learning, with Java and Python*, Chapter 16

Lecturer: Irfan Subakti — yifana@gmail.com

Every chapter so far solved one kind of problem. Real questions do not come sorted by chapter. Today one question — how large a water tank does a durian orchard need for the dry season? — is answered by a simulation assembled from the methods of the whole course, each part checked against a second method before it is trusted, and the whole repeated over hundreds of possible seasons. Book: Chapter 16, Sections 16.1–16.11.

What This Meeting Covers

1. The problem — a tank that ran dry in September.

2. The plan — a daily simulation built from checked parts.

3. The parts — tank, pump, weather, storm, drip line and sensor, each by two methods.

4. Many seasons — Monte Carlo in parallel, reproducibly.

5. The decision — golden-section search on the yearly cost, and the newsvendor rule that confirms it.

The programs exist in Java and Python and print identical results; SciPy recomputes every part. Book: Chapter 16.

Pak Hadi's Dry Season

A fruit and durian orchard east of **Jember**: sixty durian trees in full bearing. The fruit depends on getting through **June to October** without drought stress.

Water: a well with an electric pump lifting into a **13,000-litre tank**, a 120 m^2 shed roof draining into it, and a gravity drip line to every tree each morning until 08:30 at the latest.

Last September the tank was nearly empty every morning for three weeks; the harvest was smaller. A dealer sells tanks at about **Rp 1,500 per litre**. How many litres?

No formula answers this: rain is random, the well slows, the trees' needs grow, the tank's shape sets the pressure. Each is a small problem an earlier chapter can solve. Book: Section 16.1.

The Plan

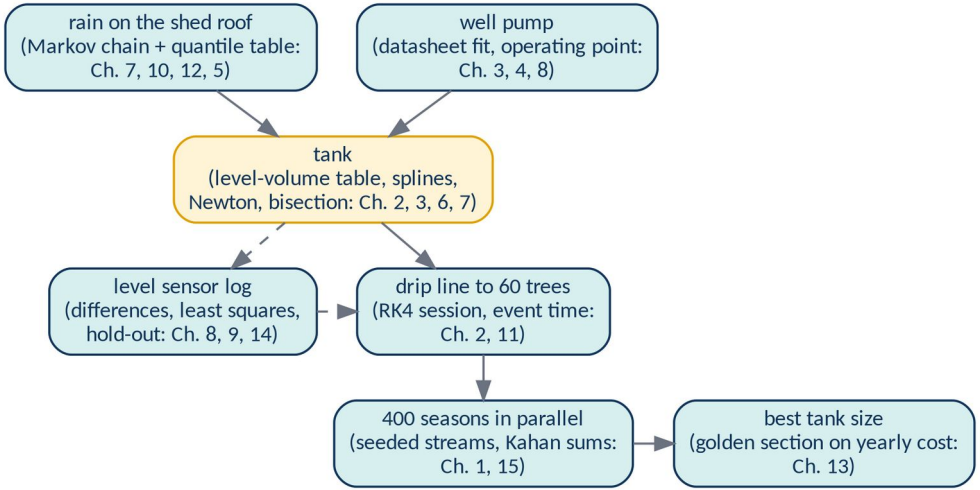


Figure 16.1 The orchard's water system as a simulation: every part is computed by methods of earlier chapters, the parts meet in the tank, and many simulated seasons feed the choice of tank size.

One day at a time: decide the weather, add roof rain, run the morning drip session, record whether the trees got their ration, pump in the evening.

One season is one possible future. Repeat for hundreds of seasons; choose the tank that is good across them.

*Figure 16.1: the parts meet in the tank. The discipline that makes a simulation trustworthy is to build it from **checked parts**: every part computed two ways before any season runs. Book: Section 16.2.*

Every Chapter Has a Job

Table 16.1 Where each chapter's methods work in the orchard simulation.

Part	Question	Methods (chapters)
Tank	Volume at a level, level for a volume?	spline, tridiagonal LU, Newton, bisection (7, 6, 3, 2)
Pump	Delivery when lifting to the tank?	least squares, elimination, Newton (8, 4, 3)
Weather	How often and how much rain?	Markov chain, power method, Gauss–Seidel (12, 5)
Storm record	Rain from one storm?	trapezoid and Simpson (10)
Drip line	How long does the ration take?	Runge–Kutta, regula falsi (11, 2)
Level sensor	Does the drip model match?	differences, least squares, held-back data (9, 8, 14)
Seasons	Many dry seasons?	seeded streams, parallel workers, compensated sums (1, 15)
Decision	Cheapest tank per year?	golden-section search (13)

Table 16.1 is the course on one page. The shared file `toolbox.py` / `Toolbox.java` collects the methods of Chapters 2 to 15 that the simulation reuses. Book: Section 16.2.

The Tank and the Pump

The tank maker's bucket table gives volume against level for 13,000 L; other sizes scale it:

$$h_C(V) = h\left(\frac{V}{C/13000}\right) \tag{16.1}$$

The pump datasheet, fitted by least squares, and the operating point where pump head meets system head:

$$H_p(Q) = a + bQ + cQ^2 \tag{16.2}$$

$$g(Q) = H_p(Q) - H_s(Q) = 0 \tag{16.3}$$

$$E = \frac{1000}{60\,Q^*}P = \frac{1000}{60 \times 59.752} \times 0.8 = 0.2231 \text{ kWh per 1000 liters} \tag{16.4}$$

Equation (16.1): a taller tank of the same shape. Equation (16.2): a quadratic fitted to the datasheet. Equation (16.3) is a root-finding problem, solved by Newton. Equation (16.4): at 0.8 kW, a thousand litres cost 0.2231 kWh, about Rp 322. Book: Sections 16.3.1–16.3.2.

The Weather as a Markov Chain

Three kinds of day — dry, light rain, heavy rain — with tomorrow depending on today (rows add to 1):

$$P = \begin{pmatrix} 0.88 & 0.10 & 0.02 \\ 0.60 & 0.32 & 0.08 \\ 0.50 & 0.35 & 0.15 \end{pmatrix} \tag{16.5}$$

$$\pi P = \pi, \pi_1 + \pi_2 + \pi_3 = 1 \tag{16.6}$$

Equation (16.5) uses the row convention, unlike Chapter 12. Equation (16.6): the long-run share of each kind of day, by the power method and by Gauss–Seidel. Book: Section 16.3.3.

The Drip Line and the Daily Balance

The outflow depends on the water level above the drippers, plus the tank's height Z on its stand:

$$\frac{dV}{dt} = -K\sqrt{\frac{h(V)}{100} + Z}\,, \quad K = 12\,\text{L/min per}\sqrt{\text{m}} \tag{16.7}$$

Each day d: rain r_d (mm) on the roof, the morning ration R_d, the evening pump volume W_d:

$$\begin{aligned} V' &= \min(C,\ V_d + 102\,r_d), \\ V'' &= V' - G_d, \qquad G_d = \text{drip session from } V' \text{ for } R_d \ (0 \text{ if } r_d \geq 10), \\ V_{d+1} &= V'' + \min(C - V'',\ W_d), \end{aligned} \tag{16.8}$$

Equation (16.7) is Chapter 11's tank law; RK4 steps it, and regula falsi finds the moment the ration is delivered. Equation (16.8): the roof catches 102 litres per millimetre of rain, a rainy day of 10 mm or more needs no watering, and anything above the capacity overflows. Book: Sections 16.3.4–16.3.5.

Running It: Every Part, Two Ways

```
1. Tank: level for 6,500 liters (Chapters 2, 3, 6, 7)
   inverse spline h(V)           115.0174 cm
   Newton on V(h) = 6500         115.0144 cm in 4 steps
   bisection on V(h) = 6500      115.0144 cm
   straight lines through table  114.7929 cm
2. Pump: datasheet fit and operating point (Chapters 3, 4, 8)
   head = 31.9958 -0.023155 Q -0.00411310 Q^2  (m, Q in L/min)
   largest misfit on the datasheet 0.087 m
   operating point 59.752 L/min at 15.927 m
   1000 liters: 0.2231 kWh, Rp 322
3. Weather: long-run share of dry, light and heavy days (5, 7, 10, 12)
   power method, 40 steps   0.828813  0.138638  0.032550
   Gauss-Seidel, 13 sweeps   0.828813  0.138638  0.032550
```

Inverse spline and Newton on the forward spline differ by 0.03 mm; straight lines by 2 mm — harmless. The pump fit misses the datasheet by at most 9 cm of head. Power method and Gauss–Seidel agree to six digits: 83% dry days. The storm integrates to 38.9 mm by the trapezoid rule and 39.1 mm by Simpson's. Book: Section 16.5.1.

Running It: the Drip Session

```
5. Drip session for 2,520 liters (Chapters 2, 11)
   start   13000 L, euler dt 10.0:  2520.00 L in 114.5825 min
   start   13000 L, rk4   dt 10.0:  2520.00 L in 114.8071 min
   start   13000 L, rk4   dt 0.5:  2520.00 L in 114.8071 min
   start    4000 L, rk4   dt 10.0:  2520.00 L in 147.2514 min
   start    3000 L, rk4   dt 10.0:  2450.54 L in 150.0000 min
```

Selected lines. From a full tank the ration takes 114.8 minutes; from 4000 L, 147.3 minutes, just inside the 150-minute window; from 3000 L the window closes 70 litres short, with 550 litres still in the tank — the pressure is too low. RK4 with 10-minute steps matches 0.5-minute steps to every digit; Euler is off by a quarter of a minute, first order as Chapter 11 predicts. Book: Section 16.5.1.

The Drip Model Against the Sensor

```
46 readings, 0 to 90 min, level 175.5 -> 151.0 cm
step  2 min: 44 rates, K (first half) 11.819, test RMS 6.839 L/min, K (all) 11.856
step 10 min: 36 rates, K (first half) 11.910, test RMS 1.253 L/min, K (all) 11.899
model value K = 12.000 L/min per sqrt(m)
```

`sensor.py` differentiates the logged level (Chapter 9), converts to litres per minute, and fits K by least squares on the first half, testing on the second (Chapters 8 and 14). With 2-minute differences the noise dominates, as Chapter 9 warned; with 10-minute differences the held-back error drops to 1.25 L/min and K comes out at 11.9 — within 1% of the model's 12. Book: Section 16.5.2.

Many Seasons: Monte Carlo

$$\hat{X} = \frac{1}{N}\sum_{s=1}^{N} X_s, \mathrm{SE}(\hat{X}) = \frac{\sigma_X}{\sqrt{N}} \tag{16.9}$$

Listing 16.1 From `montecarlo.py`: each season is seeded by its own number, so the results do not depend on the number of workers.

```
def chunk(job):
    capacity, first, count = job
    rows = []
    for i in range(first, first + count):
        tot = o.season(SEED0 + i, capacity)
        rows.append([tot[k] for k in KEYS])
    return rows
```

Equation (16.9): the Monte Carlo error falls only like $1/\sqrt{N}$ — four times the seasons for half the error. Listing 16.1: seeding by season number, not by clock or thread, is Chapter 15's reproducibility rule; the totals are summed with Kahan's method. Book: Sections 16.3.6 and 16.4.

Running It: 13,000 Against 30,000 Litres

```
tank 13000 L, 400 seasons:
  rain caught     35870 L, overflow     20496 L
  pumped         258005 L, electricity 57.6 kWh = Rp 83192
  trees got      280146 L, missed 14166 L on 26.2 days on average
  seasons with any shortage: 390 of 400; 9 in 10 miss less than 24446 L
tank 30000 L, 400 seasons:
  rain caught     35870 L, overflow     19543 L
  pumped         258745 L, electricity 57.7 kWh = Rp 83431
  trees got      292307 L, missed 2005 L on 5.8 days on average
  seasons with any shortage: 145 of 400; 9 in 10 miss less than 7446 L
```

The roof water, 35,870 L on average, agrees with the hand estimate of 35,757 within the Monte Carlo error — the first check of the whole. With the present tank, ***390 of 400*** *seasons run short. With 30,000 L, 145 seasons do, and the average shortage falls from 14,166 to 2005 litres. The electricity hardly changes: the water the trees drink has to be pumped anyway. Book: Section 16.5.3.*

One Season, Two Tanks

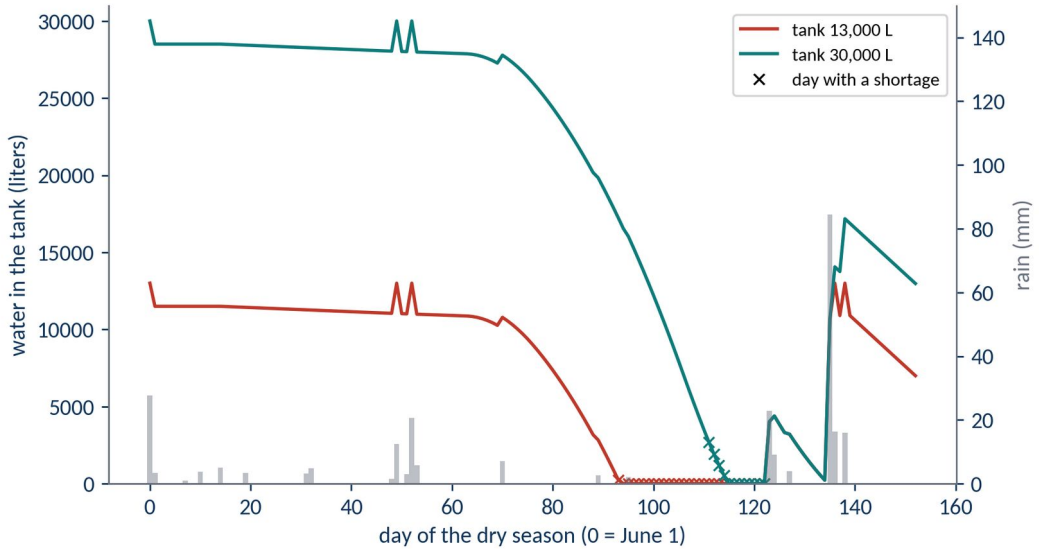


Figure 16.2 One simulated dry season (the same weather) with a 13,000-liter and a 30,000-liter tank: water in the tank after the morning watering, days on which the trees got less than their ration (crosses), and daily rain (bars, right axis).

Figure 16.2: until early August the well keeps up. Then both tanks drain at about 900 litres a day. The small tank is empty by 2 September and the trees go short every day for three weeks — exactly Pak Hadi's experience last year. The large tank lasts until a short dry spell before the first October rains. Book: Section 16.5.3.

The Decision: Cost per Year

$$\text{cost}(C) = a\,C + e\,E[\text{kWh}] + p\,E[D(C)] \tag{16.10}$$

$a = 150$ rupiah per litre of capacity per year (Rp 1,500 over ten years), $e = 1445$ rupiah per kWh, $p = 500$ rupiah per litre the trees missed — Pak Hadi's own estimate of lost fruit.

At the best size, one more litre of tank must be worth exactly what it saves — the **newsvendor rule**:

$$a = p \cdot \Pr[D(C^*) > 0] \;\Leftrightarrow\; \Pr[\text{shortage with } C^*] = \frac{a}{p} = \frac{150}{500} = 0.30 \tag{16.11}$$

Equation (16.10): tank cost rises with C, missed water falls. The expectations come from the simulation, so each evaluation is a Monte Carlo run with the same seeds for every size — common random numbers, so that the comparison is not luck. Equation (16.11) predicts the answer's shortage probability before we compute it. Book: Section 16.3.7.

Running It: Golden-Section Search on the Cost

```
200 seasons per size; yearly cost in thousand rupiah
  tank (L)  P(short)   tank  power  missed   total
     25279     0.590   3792   83.4    2082    5958
     34721     0.175   5208   83.4     347    5639
     40557     0.060   6084   83.4      64    6231
     31115     0.315   4667   83.4     804    5555
...
     31564     0.305   4735   83.4     734    5552
best tank about 31602 L, yearly cost 5552 thousand; P(short) there 0.305
newsvendor check: P(short) should be near 0.300
```

*Twelve capacities tried (rows between are left out here). About **31,600 litres**, 5.55 million rupiah a year: 4.7 million for the tank, 0.08 million electricity, 0.7 million lost fruit. The present tank would lose about 7.1 million of fruit, 9 million in all. The shortage probability at the optimum, 0.305, matches the newsvendor rule's 0.30: theory and simulation confirm each other. Book: Section 16.5.4.*

A Flat Optimum

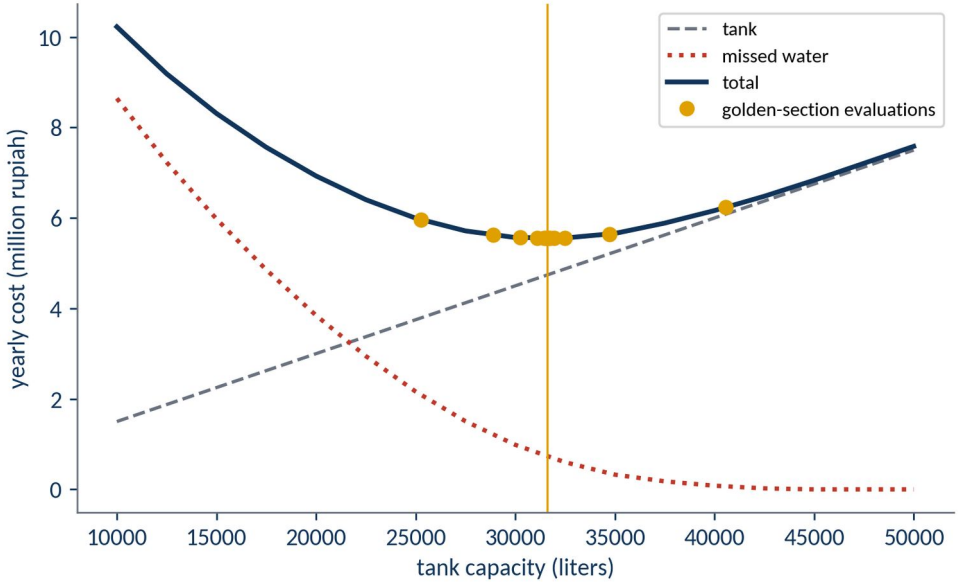


Figure 16.3 Yearly cost against tank capacity from 200 simulated seasons per size: tank cost, value of missed water, and total, with the capacities tried by the golden-section search and the chosen size.

KEY INSIGHT. The total cost is flat from about 29,000 to 34,000 litres. Report “about 30,000 litres”, not 31,602: the data — a five-number well table, an estimated fruit price — do not have five digits. Pak Hadi can buy a standard size in that range and lose almost nothing.

Figure 16.3: the tank line rises steadily, the missed-water curve falls steeply and then flattens, and the total has a broad valley. Book: Section 16.5.4.

Comparing with SciPy

```
tank: level for 6500 L 115.0174 cm (ours 115.0174)
pump: polyfit 31.9958 -0.023155 Q -0.00411310 Q^2
      operating point 59.751951 L/min (ours 59.751951)
weather: eigenvector for eigenvalue 1 0.828813  0.138638  0.032550
storm: scipy simpson 39.1111 mm
drip: solve_ivp reaches 2520 L after 114.8071 min (ours 114.8071)
```

`CubicSpline`, `polyfit`, `brentq`, `eig`, `simpson` and `solve_ivp` with an event function recompute every part and agree to every printed digit. Book: Section 16.6.

Making It Better, and Using It for Real

Leave room for the rain. More than half of the roof water overflows because the evening refill keeps the tank full; refill only to 90% in August and September.

Test the sensitivity. Change the well's September yield, the weather probabilities and the value of lost fruit by 10% and rerun `design.py`.

Another orchard, a household, a school or a small clinic with a rainwater tank can use the model by changing named constants and data files. For a household buying water from a truck at Rp 30 per litre, $a/p = 5$ *and no large tank pays for itself on that account alone — the same rule, a different answer. Book: Sections 16.7–16.8.*

Numerical Methods in Practice: Profitina

Profitina's AI Visibility platform is, like the orchard, a **system of parts**: a scan measures many signals, each computed by its own method and checked on its own; they are normalized and combined by weights into a score.

The score is followed in a weekly trend, with a scheduler that re-scans each brand at a cadence set by its subscription tier.

"Check every part against an independent method before trusting the whole" is the discipline of a **release gate**: well over a thousand automated tests must pass, and a release counts as successful only when the deployed version answers its health check.

The exact signals and weights are Profitina's own; the discipline is this chapter's. As in every chapter, the numbers advise and the people decide. Book: Section 16.9.

Common Mistakes

Assembling before checking. A wrong sign in the pump fit still prints plausible numbers.

Different weather for each option. Use the same seeds for every size, or you compare luck.

Seeding from the clock or the thread. Seed each season from its number.

Trusting digits the data do not have. Report the flat range, not the exact optimum.

Averages alone. Report the probability of any shortage and a high percentile too.

Book: Section 16.10.

Summary

1. A real question was answered by a simulation built from the methods of the whole course (Table 16.1, Figure 16.1).

2. Every part was checked by a second method: tank (Equation (16.1)), pump (Equation (16.2), Equation (16.3)), weather (Equation (16.5), Equation (16.6)), drip (Equation (16.7)).

3. The daily balance (Equation (16.8)) ran 153 days per season, 400 seasons in parallel, reproducibly; the error falls like $1/\sqrt{N}$ (Equation (16.9)).

4. The present tank runs short in almost every season; golden-section search on Equation (16.10) chose about 31,600 litres, flat from 29,000 to 34,000.

5. The shortage probability there, 0.305, matches the newsvendor rule (Equation (16.11)); Java, Python and SciPy agree.

From Bu Wati's receipts in Chapter 1 to Pak Hadi's tank, the same questions ran through every method: how wrong can this be, how fast does it converge, and how do we know? Book: Section 16.11.

The Exercises

Find the stationary weather distribution by hand from the balance equations and the sum condition, and check it against the program.

Stop the evening refill at 90% of capacity in August and September and rerun `montecarlo.py` for 30,000 litres: what happens to overflow and missed water?

Without running anything, use the newsvendor rule for a fruit value of Rp 300 or Rp 1,000 per litre.

Another exercise asks how many seasons give the shortage probability with a standard error of 0.01, and how long that takes in each language. The solutions are in the book. Book: Sections 16.12–16.13.