py-mathx-lab Documentation

Release 1.1.0.dev11

Walter Weinmann

Dec 27, 2025

CONTENTS

Small, reproducible math experiments implemented in Python.

- **Audience:** curious engineers, students, and researchers
- **Idea:** each experiment is a self-contained runnable module with a short write-up
- **Goal:** a growing, searchable "lab notebook" of experiments

CHAPTER ONE

START HERE

- *Mathematical experimentation* - what “experiments” in mathematics mean and how to read this repo
- *Experiments Gallery* - experiment gallery (IDs, tags, how to run)
- *Valid Tags* - central directory of valid tags for experiments
- *Background* - mathematical background for experiments
- *Getting started* - install, setup, and run your first experiment
- *Development* - Makefile workflow, CI, coding conventions
- *References* - bibliography and reading list
- *PDF download* - download the PDF version of these docs

CHAPTER

TWO

RUN ONE EXPERIMENT

```
make uv-check
make venv
make install-dev
make run EXP=e001 ARGS="--out out/e001 --seed 1"
```

CHAPTER

THREE

LATEST

- – **E011** - *E011: Heuristic rarity of Mersenne primes*

3.1 Mathematical experimentation

Mathematical experimentation is the practice of using examples, computation, and visualization to discover structure, to generate conjectures, find counterexamples, estimate quantities, and build intuition that later supports (or refutes) formal arguments.

It is *not* "proof by computer output". Instead, experiments are a disciplined way to ask better questions and to stress-test ideas-especially now that modern computers make it easy to explore large search spaces, high-precision numerics, and rich visualizations.

This repository, **py-mathx-lab**, is a small "lab notebook" of such experiments: compact, reproducible, and readable.

3.1.1 What counts as an "experiment" in mathematics?

An experiment is a finite procedure that produces evidence about a mathematical claim or object. Typical outcomes:

- **Conjecture generation**: patterns suggest statements that might be true (or false).
- **Counterexample search**: systematic exploration tries to break a hypothesis early.
- **Quantitative exploration**: estimate constants, rates, limits, or distributions.
- **Model checking**: validate (or invalidate) approximations and heuristics.
- **Visualization**: reveal structure that is hard to see symbolically.

The modern viewpoint-where computation is a genuine part of mathematical discovery-is widely discussed under the name *experimental mathematics* ([BB05, BBC04, Bor05]). Some authors emphasize "plausible reasoning" supported by computation, paired with careful verification and eventual proof ([Bor09, BB08]). Other texts take a more problem-driven, exploratory style aimed at students and engineers ([LCD+03]), or present computation as a tool to formulate concrete research problems ([Arn15]).

3.1.2 Why computers change the game

Computers do not replace mathematical thinking-but they expand what is feasible to *inspect*:

- **Scale**: enumerate millions of cases (to find the first failure or build confidence).
- **Precision**: compute with high-precision floats or exact rationals/integers to avoid roundoff illusions.
- **Multiple lenses**: combine numerics, exact arithmetic, symbolic manipulation, and plotting.
- **Search**: automate discovery (parameter sweeps, optimization, random sampling, heuristics).
- **Reproducibility**: re-run the same pipeline with fixed seeds and pinned dependencies.

Used well, computation turns “I wonder if…” into “here is evidence, here are edge cases, and here is what we should prove next”.

3.1.3 Experiments vs. proofs

A proof is the end of the story; an experiment is often the beginning.

Experiments are excellent for:

- **disproving** statements (one counterexample ends it),
- identifying **what is actually true** (after a naive conjecture fails),
- suggesting **lemmas** and **invariants** that make a proof possible.

But experiments can also mislead. Common failure modes:

- floating point error and catastrophic cancellation,
- plotting artifacts,
- “pattern matching” based on too few samples,
- unintentional selection bias (“I tried the cases that worked”).

The goal is to use experiments to *reduce uncertainty*, not to hide it.

3.1.4 Targets for py-mathx-lab

py-mathx-lab aims to be a practical, long-lived collection of experiments with shared conventions:

1. **Reproducible runs**
   Each experiment is runnable as a module, writes results to a single output directory, and (when relevant) uses a fixed seed.
2. **Readable code**
   The code should be short, well-typed, and structured so readers can modify it.
3. **Useful artifacts**
   Each experiment should generate at least one of:
   - a figure
   - a table / summary statistics
   - a counterexample / witness object
   - a short narrative explaining what was learned
4. **Clear boundaries**
   An experiment write-up should state:
   - what is being tested,
   - what counts as “success” or “failure”,
   - what might invalidate the result (precision limits, domain constraints, runtime limits).
5. **Traceability**
   Each page includes references to books/papers that motivated the work or explain the background.

3.1.5 Repository conventions for experiments

An experiment page should usually include:

- **Goal** (one paragraph)
- **How to run** (a command that works from the repo root)
- **Parameters** (including defaults and ranges, if swept)

- **Results** (figures/tables and a short interpretation)
- **Notes / pitfalls** (numerical caveats, surprising behavior)
- **References** (bibliography keys)

In code, prefer:

- deterministic outputs (fixed seeds, stable sorting),
- explicit configuration objects / CLI arguments,
- sanity checks (dimension checks, bounds checks, invariants),
- cross-checks (e.g., float vs. exact, two independent formulas).

3.1.6 Examples of good "experiment themes"

The experiment format supports many domains, for example:

- **Numerical analysis**: error landscapes, stability regions, conditioning, Monte Carlo integration.
- **Number theory**: continued fractions and convergents, integer sequences, modular patterns, primality heuristics.
- **Geometry/topology**: random point clouds, curvature approximations, combinatorial invariants.
- **Optimization/probability**: stochastic search behavior, concentration phenomena, empirical distributions.

A concrete example of a rich, experiment-friendly topic is continued fractions, where it is natural to compute and plot convergents, partial quotient statistics, and periodicity phenomena ([BvdPSZ14]).

Next steps: start with *Getting started*, browse the *Valid Tags* directory, and then browse the *Experiments Gallery* gallery.

3.2 Valid Tags

This page defines the allowed tags for experiments in **py-mathx-lab**. Tags are used in the *Experiments Gallery* and individual experiment pages to categorize content.

3.2.1 Primary Tags (Domains)

These represent the broad mathematical area of the experiment.

Tag	Description
`analysis`	Calculus, real/complex analysis, limits, and approximation.
`number-theory`	Properties of integers, divisibility, and prime numbers.
`conjecture-generation`	Patterns suggest statements that might be true (or false).
`counterexample-search`	Systematic exploration tries to break a hypothesis early.
`quantitative-exploration`	Estimate constants, rates, limits, or distributions.
`model-checking`	Validate (or invalidate) approximations and heuristics.
`visualization`	Reveal structure that is hard to see symbolically.

3.2.2 Secondary Tags (Topics & Methods)

These provide more specific detail about the techniques or sub-topics involved.

Tag	Description
`classification`	Grouping objects into classes based on shared properties.
`exploration`	Open-ended search for patterns or properties.
`numerics`	Heavy use of floating-point or high-precision computation.
`open-problems`	Related to famous unproven conjectures.
`optimization`	Finding maxima, minima, or best-fit parameters.
`perfect`	Related specifically to perfect, abundant, or deficient numbers.
`search`	Systematic search through a large state space.
`sigma`	Related to the sum-of-divisors function $\sigma(n)$.
`taylor`	Related to Taylor series and their approximations.

3.2.3 Usage

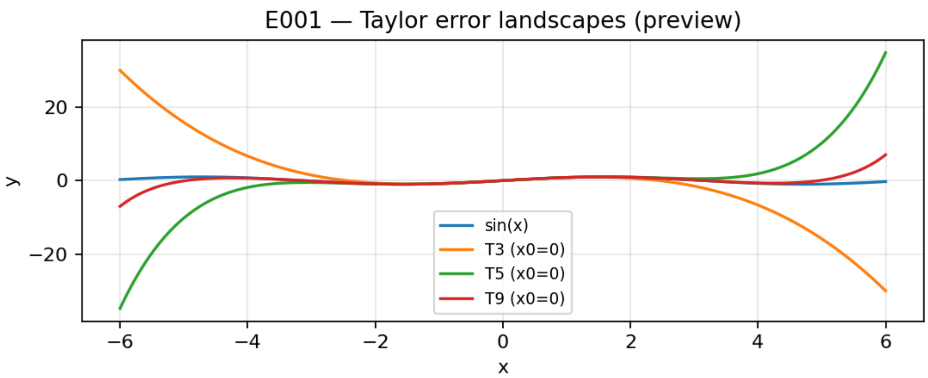
When adding a new experiment:

1. Choose at least one **Primary Tag** (Domain or Type).
2. Choose one or more **Secondary Tags** (Topics & Methods).
3. Add them to the `**Tags:**` line in your `.md` file.
4. Update the *Experiments Gallery* using the corresponding CSS classes (`tag-primary` for primary tags, `tag-secondary` for secondary tags).

3.3 Experiments Gallery

A compact, image-first overview of the experiments in **py-mathx-lab**.

3.3.1 E001 — Taylor Error Landscapes



Tags: analysis, quantitative-exploration, visualization, numerics, taylor (see *Valid Tags*)

Goal

Build intuition for Taylor truncation error by visualizing the absolute error landscape $|\sin(x) - T_n(x; x_0)|$ over a domain while varying:

- the polynomial degree n,
- the expansion center x_0.

Background (quick refresher)

If you want a short mathematical recap first, read: *Taylor series refresher*.

Research question

How does the approximation error of Taylor polynomials for $\sin(x)$ depend on the polynomial degree n and the expansion center x_0, over a fixed domain of x?

Concretely: for a chosen grid of (n, x_0), what does the error landscape $E_{n,x_0}(x) = |\sin(x) - T_n(x; x_0)|$ look like, and where do numerical artifacts start to dominate the truncation error?

Why this qualifies as a mathematical experiment

This page is not just a worked example or a derivation — it is an *experiment* in the sense of **experimental mathematics**: a finite, reproducible procedure that produces **evidence** about how a mathematical object behaves.

For E001, the object is the family of Taylor polynomials $T_n(x; x_0)$ for $\sin(x)$, and the observable is the error function $E_{n,x_0}(x) = |\sin(x) - T_n(x; x_0)|$. The experiment qualifies because it:

- **Explores a parameter space:** it varies degree n and center x_0 and inspects how the entire error landscape changes.
- **Generates testable conjectures:** e.g. "there is a widening low-error region around x_0 as n increases" and "improvement is not uniform across a fixed interval".
- **Searches for failure modes / edge cases:** large $|x - x_0|$, high degrees, and wide domains can expose numerical artifacts that *look* like truncation error but are actually floating-point limitations.
- **Produces artifacts that can be checked independently:** plots and parameter snapshots make it easy to compare runs, reproduce the same conditions, and verify that an observed pattern is not accidental.

The goal is to turn "Taylor series should be good near x_0" into *structured evidence* about **where** and **how** the approximation is good (or bad), which then informs what one would try to prove or bound formally.

- How does the truncation error $|\sin(x) - T_n(x; x_0)|$ behave as we move away from the center x_0?
- How many terms are needed to achieve a specific precision (e.g., 10^{-6}) over a fixed interval?
- Does increasing the degree n always improve the result everywhere in the domain?

Experiment design

- **Target function:** $\sin(x)$
- **Evaluation:** $x \in [-2\pi, 2\pi]$ (default)
- **Parameters:**
  - `degrees`: $\{1, 3, 5, 7, 9\}$
  - `centers`: $\{0, \pi/2, \pi\}$
- **Outputs:**
  - Plot of $f(x)$ vs $T_n(x)$
  - Semi-log plot of $|f(x) - T_n(x)|$

How to run

```
make run EXP=e001 ARGS="--seed 1"
```

Artifacts are written under `out/e001/` (figures, parameters, and a short `report.md`).

What to expect

Qualitatively (and this is what the plots should confirm):

- near x_0, the higher degree reduces the error quickly,
- away from x_0, the approximation can degrade even for higher degrees,
- changing x_0 shifts the "low-error region".

Results

After running the experiment, include (or regenerate) the figures in the documentation. The canonical output location is `out/e001/`. For publishing, copy one representative "hero" image into `docs/_static/experiments/` (see "Gallery images" below).

Notes / pitfalls

- Use **log-scale** for absolute error plots to see the full dynamic range.
- Be careful interpreting relative error near zeros of $\sin(x)$.
- Huge domains and high degrees can expose floating-point artifacts that are not truncation error.

Extensions

- **Alternative functions:** Repeat the experiment for $exp(x)$ or $1/(1-x)$.
- **Relative error:** Plot $|(f - T_n)/f|$ instead of absolute error.
- **Automatic degree selection:** Find the minimal n such that error $< \epsilon$ on a given interval.

Gallery images (recommended)

To keep the experiment gallery attractive and stable:

1. run the experiment locally,
2. pick one representative output figure,
3. copy it into the repo under:

   `docs/_static/experiments/e001_hero.png`

This allows the docs to show thumbnails without depending on generated `out/` artifacts.

References

See *References*. [BB05, Bor05, BB08]

3.3.2 E002: Even Perfect Numbers — Generator and Growth

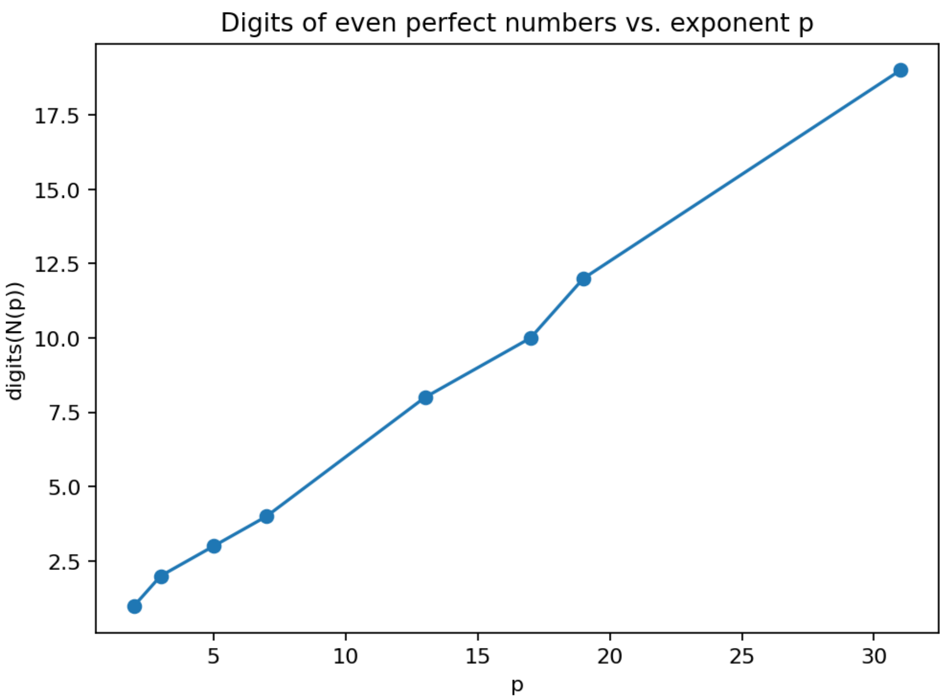
Tags: number-theory, quantitative-exploration, visualization, numerics (see *Valid Tags*)

Highlights

- Generate even perfect numbers from known Mersenne exponents p.
- Plot digits and bit-length growth vs. p.
- Test the approximation $\log_{10}(N(p)) \approx 2p \log_{10}(2)$.

Goal

Generate **even perfect numbers** from known Mersenne prime exponents p and visualize how fast the numbers grow. Measure growth via **digit count**, **bit length**, and simple logarithmic approximations.



Research question

For

$$N(p) = 2^{p-1}(2^p - 1),$$

how do:

- $\mathrm{digits}(N)$,
- $\mathrm{bit\_length}(N)$,
- $\log_{10}(N)$

scale with the exponent p?

How accurate is the approximation

$$\log_{10}(N(p)) \approx 2p \log_{10}(2)?$$

Why this qualifies as a mathematical experiment

The Euclid–Euler theorem tells us exactly what even perfect numbers look like, but it does not directly convey how quickly the objects become astronomically large. This experiment uses computation and visualization to build quantitative intuition and test simple asymptotic approximations.

Experiment design

Inputs

- A curated list of known Mersenne prime exponents, e.g. $p \in \{2, 3, 5, 7, 13, 17, 19, 31, \dots\}$.

Observables

For each exponent p:

- $N(p)$ as a Python integer
- $\mathrm{digits}(N(p))$

- bit_length($N(p)$)
- approximation error:

$$\Delta(p) = \log_{10}(N(p)) - 2p\log_{10}(2)$$

Plots

- p vs. digits
- p vs. bit length
- p vs. $\Delta(p)$

How to run

From the repo root:

```
make run EXP=e002
```

or:

```
uv run python -m mathxlab.experiments.e002
```

Notes / pitfalls

- Avoid converting huge integers to decimal strings repeatedly in tight loops; compute digits using logs where possible.
- Use `int.bit_length()` for stable bit length (fast and exact).
- Keep the exponent list modest for fast docs builds; this is a growth experiment, not a search for new Mersenne primes.

Extensions

- Compare growth to 2^{2p} and quantify the relative gap.
- Add a "human scale" axis: compare digits($N(p)$) to common benchmarks (atoms in the observable universe, etc.).
- Pull the exponent list from a small data file so the experiment can be updated without code changes.

References

See *References*.

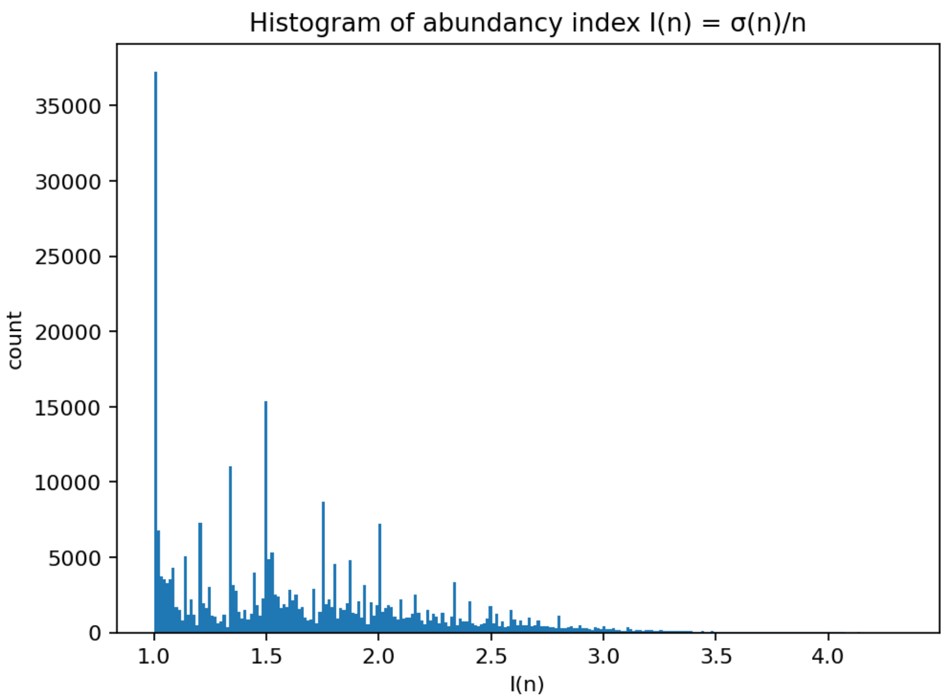
[Cald., Voi98, OEISFInc25]

3.3.3 E003: Abundancy Index Landscape

Tags: number-theory, quantitative-exploration, visualization, numerics (see *Valid Tags*)

Highlights

- Compute $\sigma(1..N)$ via a divisor-sum sieve.
- Visualize the distribution of $I(n) = \sigma(n)/n$.
- Highlight perfect numbers as the razor-thin level set $I(n) = 2$.



Goal

Visualize how **rare** perfect numbers are by plotting the distribution of the **abundancy index**

$$I(n) = \frac{\sigma(n)}{n}$$

for integers $n \le N$. Perfect numbers satisfy $I(n) = 2$.

Research question

For a given bound N:

- What does the empirical distribution of $I(n)$ look like?
- How frequently do we see values near 2?
- Where do the perfect numbers appear in the landscape?

Why this qualifies as a mathematical experiment

The divisor-sum function $\sigma(n)$ is highly structured but "spiky" and hard to intuit symbolically. Computational sweeps reveal qualitative structure (clusters, gaps, tails) and put perfect numbers into context.

Experiment design

Method: compute (1), …, (N) via a divisor-sum sieve

Use the identity "each divisor contributes to its multiples":

- initialize an array `sigma[0..N]` with zeros
- for each `d = 1..N`:
  - add `d` to `sigma[k]` for all multiples `k = d, 2d, 3d, ...`

This runs in about $O(N \log N)$ time and avoids per-number factorization.

Observables

- $I(n) = \sigma(n)/n$
- distance to perfection: $|I(n) - 2|$
- classification:
  - deficient: $I(n) < 2$
  - perfect: $I(n) = 2$
  - abundant: $I(n) > 2$

Plots

- histogram of $I(n)$ (or of $I(n) - 1$)
- scatter plot of n vs. $I(n)$ (optionally with log-scale on n)
- highlight perfect numbers

How to run

```
make run EXP=e003
```

or:

```
uv run python -m mathxlab.experiments.e003
```

Notes / pitfalls

- $I(n)$ is rational. For classification, compare integers using $\sigma(n)$ and $2n$ rather than floats.
- For plots, floats are fine, but compute the “perfect” condition as `sigma[n] == 2*n`.
- Start with $N \leq 1\,000\,000$ to keep runtime and memory reasonable.

Extensions

- Repeat for different ranges and overlay histograms.
- Plot the top-k values of $I(n)$ and compare to known extremal families.
- Explore the “near misses” set (feeds directly into E006).

References

See *References*.

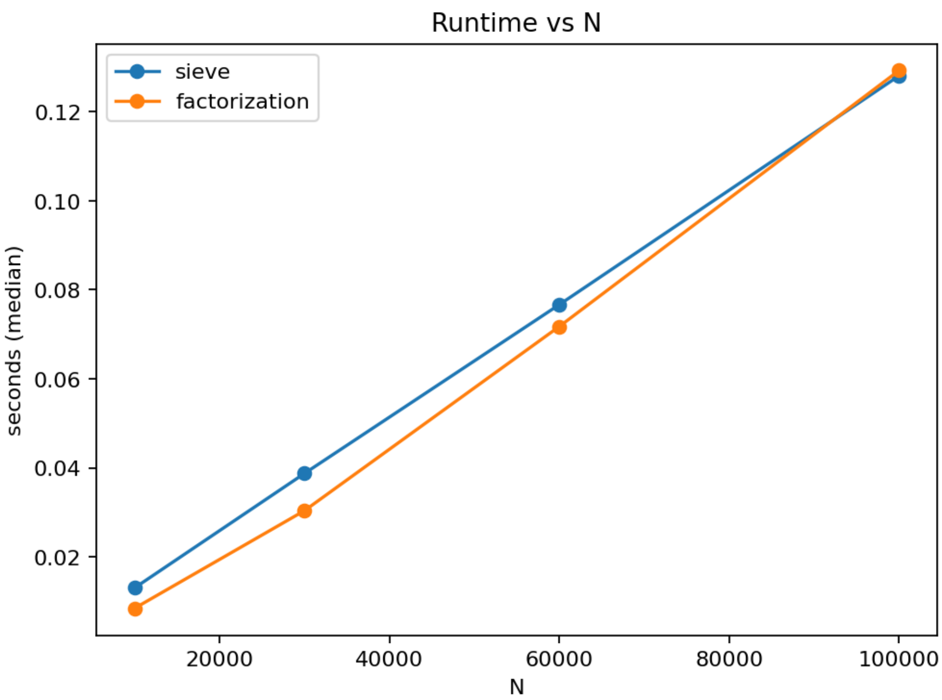
[Voi98, Wei03, OEISFInc25]

3.3.4 E004: Computing (n) at Scale — Sieve vs. Factorization

Tags: number-theory, quantitative-exploration, numerics, optimization (see *Valid Tags*)

Highlights

- Benchmark bulk sieve vs. per-number factorization.
- Find runtime crossover points as N grows.
- Validate correctness on sampled inputs.



Goal

Benchmark two practical ways to compute the sum-of-divisors function:

1. **Sieve method:** compute all $\sigma(1..N)$ in one pass.
2. **Factorization method:** compute $\sigma(n)$ per-number from the prime factorization.

Research question

For a range of bounds N:

- which approach is faster?
- where is the crossover point?
- what are the memory tradeoffs?

Why this qualifies as a mathematical experiment

Both methods are mathematically equivalent, but performance depends on constants, caching, and implementation details. This is a quantitative exploration of algorithmic behavior grounded in number-theory structure.

Experiment design

Method A: divisor-sum sieve (bulk computation)

Compute `sigma[1..N]` by adding each divisor to its multiples (as in E003).

Method B: per-number factorization

Factor each n (e.g. using a precomputed prime list up to $\sqrt{N}$) and compute:

If

$$n = \prod_{i=1}^{k} p_i^{a_i},$$

then

$$\sigma(n) = \prod_{i=1}^{k} \frac{p_i^{a_i+1} - 1}{p_i - 1}.$$

Measurements

- wall-clock runtime vs. N (multiple trials, median)
- peak memory (rough estimate acceptable for v1)
- correctness cross-check: random sample where both methods agree

Outputs

- plot: runtime vs. N for both methods
- short table: N, runtime(A), runtime(B), speedup

How to run

```
make run EXP=e004
```

or:

```
uv run python -m mathxlab.experiments.e004
```

Notes / pitfalls

- Factorization becomes expensive quickly; keep the factorization method limited to moderate N.
- Use integer arithmetic for correctness checks (`sigma[n] == 2*n` etc.).
- Report both runtime and *effective throughput* (numbers processed per second).

Extensions

- Add a third method using smallest-prime-factor (SPF) sieve for fast factorization.
- Compare pure Python vs. `numpy` arrays for Method A.
- Turn the benchmark into a reusable utility for later experiments.

References

See *References*.

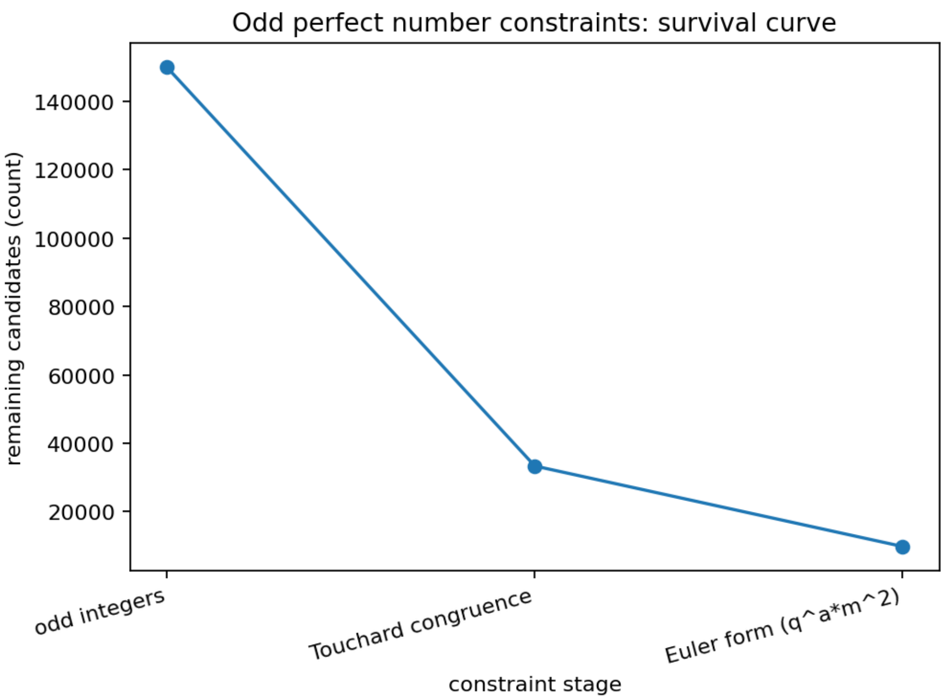
[Voi98, Wei03]

3.3.5 E005: Odd Perfect Numbers — Constraint Filter Pipeline

Tags: number-theory, counterexample-search, visualization, search (see *Valid Tags*)

Highlights

- Apply necessary constraints as staged filters on odd candidates.
- Plot survival curves (remaining candidates per constraint stage).
- Make the “open problem” constraints tangible at finite scales.



Goal

Demonstrate *why brute-force search for odd perfect numbers is unrealistic* by applying known **necessary conditions** as a step-by-step filter pipeline and visualizing how the candidate pool collapses.

Research question

Given odd integers up to a bound N:

- how many survive each constraint stage?
- which constraints are most "eliminating" in practice at small scales?
- what does a survival curve suggest about scalability?

Why this qualifies as a mathematical experiment

Odd perfect numbers are an open problem. Theorems provide constraints rather than a classification. Computation makes these constraints tangible by measuring their elimination power on finite candidate sets.

Experiment design

Candidate set

Start with odd integers $1 < n \leq N$.

Constraint stages (v1)

Use a conservative, well-known set of *necessary* conditions that can be checked mechanically:

1. **Euler form (shape constraint):** any odd perfect number must be of the form

$$n = q^{\alpha} m^2$$

where q is prime, $\gcd(q, m) = 1$, and

$$q \equiv \alpha \equiv 1 \pmod 4.$$

2. **Congruence filter (Touchard-type):** odd perfect numbers satisfy strong congruence restrictions (implemented as one or two simple congruence checks supported by the references).
3. **Small-prime structure filters:** apply a few lightweight necessary conditions (e.g., reject candidates divisible by some specific small patterns if justified by the reference set).

For each stage, record "remaining candidates".

Output

- table: stage name, remaining count, elimination percentage
- plot: survival curve (stage index vs. remaining candidates)

How to run

```
make run EXP=e005
```

or:

```
uv run python -m mathxlab.experiments.e005
```

Notes / pitfalls

- Be explicit about what is *proved* vs. what is a heuristic. Only implement conditions that are truly necessary.
- At small N, some deep constraints won't show their full strength; the goal is the *pipeline idea*, not a record bound.
- Euler-form testing requires factorization; keep N small enough for trial division (v1).

Extensions

- Add stronger bounds/constraints from the modern literature and compare elimination power.
- Replace trial division with an SPF sieve to scale the Euler-form test.
- Report not just counts but also the distribution of remaining prime-factor patterns.

References

See *References*.

[Guy04, OR14, Sto24, Voi98]

3.3.6 E006: Near Misses to Perfection

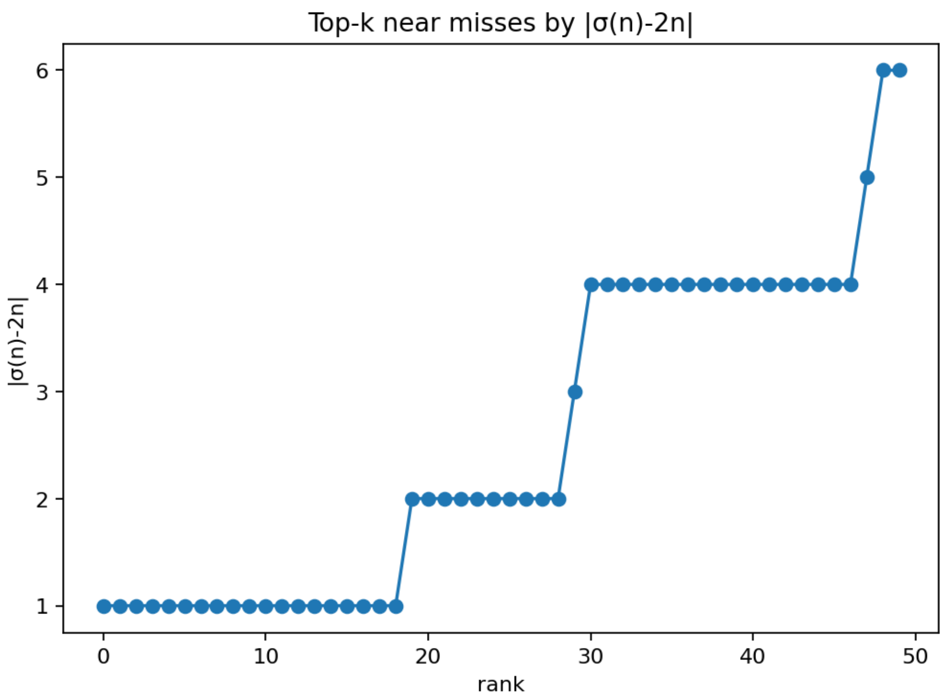
Tags: number-theory, conjecture-generation, visualization, numerics (see *Valid Tags*)

Highlights

- Search for n with $\sigma(n)$ unusually close to $2n$.
- Build leaderboards for absolute and relative deviation.
- Visualize how "near perfection" clusters by structure.

Goal

Find and visualize integers whose divisor sum is **unusually close** to the perfect condition $\sigma(n) = 2n$, without being perfect.



Research question

For integers $n \le N$, which numbers minimize:

- absolute deviation:

$$D_1(n) = |\sigma(n) - 2n|$$

- relative deviation:

$$D_2(n) = \left| \frac{\sigma(n)}{n} - 2 \right| ?$$

Do "near misses" cluster in recognizable families (highly composite, abundant, etc.)?

Why this qualifies as a mathematical experiment

The perfect condition is a sharp equality. Studying the closest failures often reveals structure and suggests new questions (e.g., which multiplicative patterns drive $\sigma(n)$ toward $2n$).

Experiment design

Computation

- Compute $\sigma(1..N)$ via the divisor-sum sieve (as in E003).
- For each n, compute $D_1(n)$ and $D_2(n)$.
- Keep the top-k smallest deviations (excluding actual perfect numbers).

Outputs

- table: top-k near misses (with n, $\sigma(n)$, D_1, D_2)
- plot: n vs. $D_2(n)$ (log-scale on D_2 often helps)
- mark perfect numbers for reference

How to run

```
make run EXP=e006
```

or:

```
uv run python -m mathxlab.experiments.e006
```

Notes / pitfalls

- Use integer comparisons to identify perfect numbers (`sigma[n] == 2*n`).
- For D_2, floats are fine for plotting, but store exact rational values for ranking when possible (e.g., compare $|\sigma(n) - 2n|$ first, then normalize for reporting).
- Choose k small (e.g. 50 or 200) so the report stays readable.

Extensions

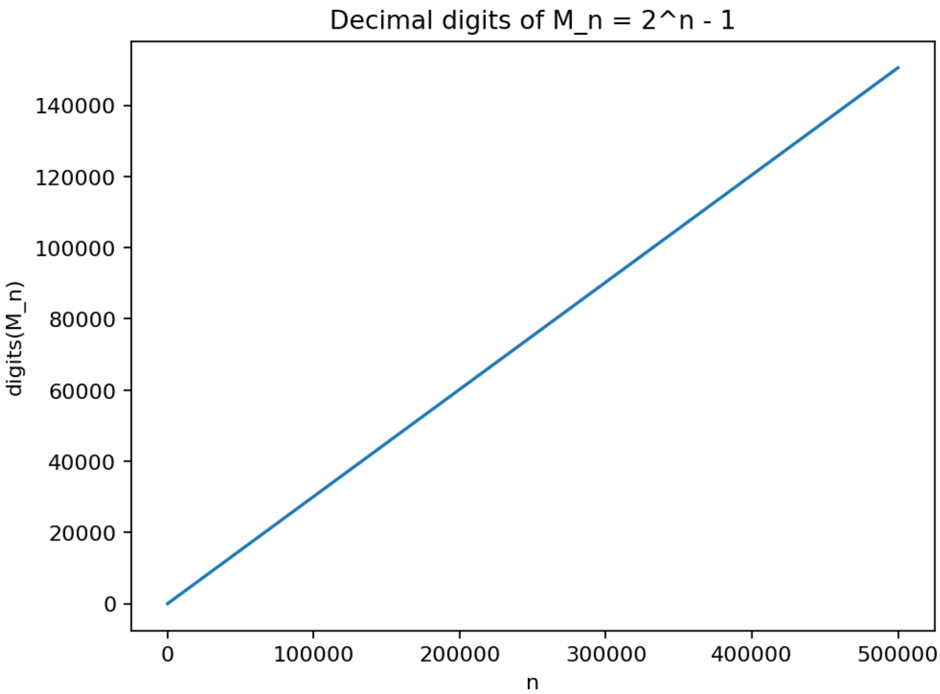
- Repeat for different N and compare stability of the "near miss" leaderboard.
- Add a second leaderboard restricted to odd n only.
- Compare near misses to known abundant/deficient classifications and prime factorizations.

References

See *References*.

[Voi98, Wei03, OEISFInc25]

3.3.7 E007: Mersenne growth (bits and digits)



Tags: number-theory, quantitative-exploration, visualization (see *Valid Tags*)

Highlights

- Visualize how quickly $M_n = 2^n - 1$ grows in **bits** and **decimal digits**.
- Compute size metrics *without* constructing huge integers.
- Provide simple rules-of-thumb for choosing feasible bounds in later experiments.

Goal

Quantify and visualize the growth of Mersenne numbers $M_n = 2^n - 1$ as n increases.

This experiment focuses on **size** (bit-length and decimal digits), because it determines what later algorithms (Lucas–Lehmer, sieving) can realistically handle on a laptop.

Background (quick refresher)

- *Mersenne numbers and primes refresher*

Research question

How fast does the size of M_n grow with n, and what simple formulas approximate:

- the number of bits of M_n
- the number of decimal digits of M_n

Why this qualifies as a mathematical experiment

- **Finite procedure:** evaluate size formulas on a range $n = 1..N$.
- **Observable(s):** bit-length and digit count as functions of n.
- **Parameter space:** vary N (and optional sampling density).
- **Outcome:** visual evidence (curves) and tables usable as planning guidance.
- **Reproducibility:** parameters written to `params.json`; figures written to `figures/`.

Experiment design

Computation

Key facts:

- $\text{bits}(2^n - 1) = n$ for $n \geq 1$.
- $\text{digits}(2^n - 1) = \lfloor n \log_{10}(2) \rfloor + 1$ for $n \geq 1$.

Compute both for a grid of n values.

Outputs

- plot: n vs. bits
- plot: n vs. digits
- table: selected n with digits (e.g., 10, 100, 1k, 10k, …)

How to run

```
make run EXP=e007
```

or:

```
uv run python -m mathxlab.experiments.e007
```

Notes / pitfalls

- Avoid converting huge M_n to decimal just to count digits. The logarithm formula is exact for the digit count.
- For very large n, use stable floating-point evaluation for $n \log_{10}(2)$ (double precision is fine up to very large n).
- This experiment is intentionally lightweight; it should run quickly.

Extensions

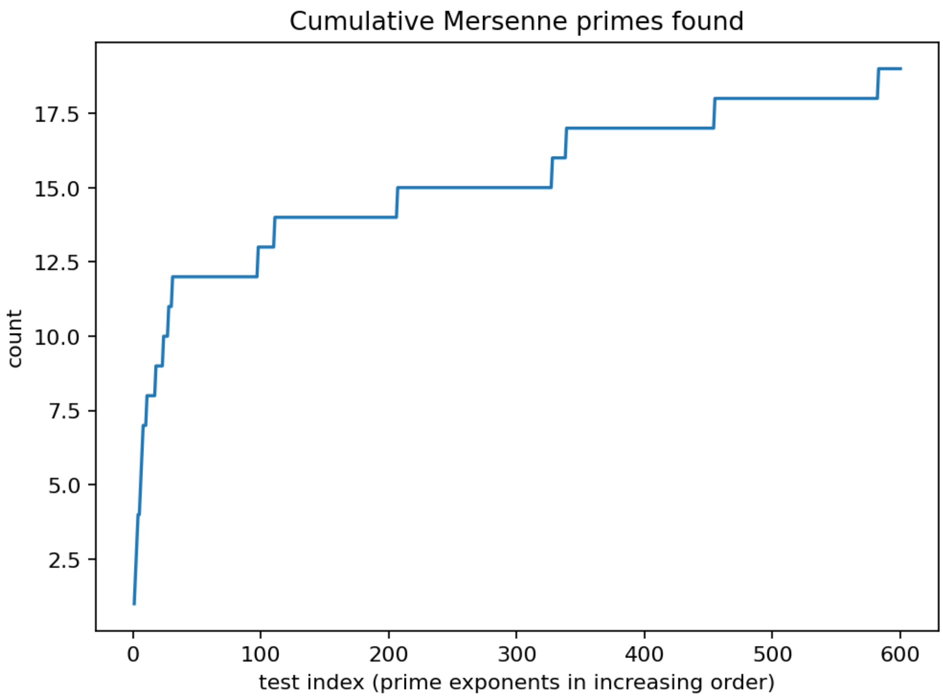
- Overlay "feasibility bands" (e.g., digits thresholds) for what your other experiments can handle.
- Compare M_n to other fast-growing families (factorials, primorials) on a log scale.

References

See *References*.

[con25b]

3.3.8 E008: Lucas–Lehmer scan (prime exponents)



Tags: number-theory, quantitative-exploration, visualization (see *Valid Tags*)

Highlights

- Run the Lucas–Lehmer test for many prime exponents p.
- Record timing and residue behavior for composite vs. prime outcomes.
- Produce a reproducible "scan report" (which p were tested, which passed).

Goal

Empirically explore Mersenne primality testing by scanning prime exponents p and applying the Lucas–Lehmer test to $M_p = 2^p - 1$.

Background (quick refresher)

- *Mersenne numbers and primes refresher*

Research question

For prime exponents $p \le P$, how does:

- runtime of the Lucas–Lehmer test scale with p?
- the distribution of final residues (mod M_p) differ between prime and composite outcomes?

Why this qualifies as a mathematical experiment

- **Finite procedure:** enumerate prime exponents up to a bound and test them.
- **Observable(s):** pass/fail, final residue, and runtime.
- **Parameter space:** vary P and optional "max tests" cap.
- **Outcome:** a dataset (tested p values) and plots/tables that suggest scaling behavior.
- **Failure modes:** naive implementations can be slow; parameters bound the search to keep runs feasible.

Experiment design

Computation

- Enumerate prime exponents p up to a bound P.
- For each p, compute $M_p = 2^p - 1$ and run Lucas–Lehmer:
  - $s_0 = 4$
  - $s_{k+1} = s_k^2 - 2 \pmod{M_p}$
  - M_p is prime iff $s_{p-2} \equiv 0 \pmod{M_p}$.
- Record (at minimum): p, pass/fail, final residue, and wall time.

Outputs

- table: scanned exponents with result and time
- plot: p vs runtime (often log-scale is helpful)
- plot: residue magnitude / patterns (optional)

How to run

```
make run EXP=e008
```

or:

```
uv run python -m mathxlab.experiments.e008
```

Notes / pitfalls

- Only run Lucas–Lehmer for **prime** p; if p is composite then M_p is composite.
- The test is defined for odd prime p; treat $p = 2$ separately ($M_2 = 3$ is prime).
- Keep bounds modest at first; the cost grows quickly with p.

Extensions

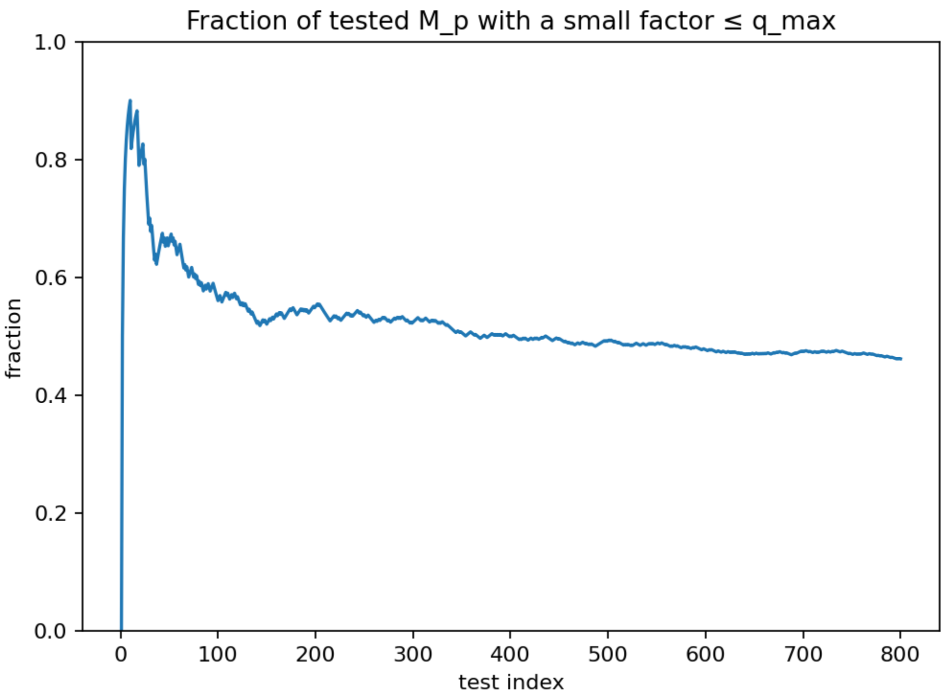
- Add a "pre-sieve" that checks for small factors before running Lucas–Lehmer.
- Compare your timings to published implementations (as a sanity check, not a competition).

References

See *References*.

[Cal21, con25a, MR24]

3.3.9 E009: Small-factor scan for M_p



Tags: number-theory, quantitative-exploration, visualization (see *Valid Tags*)

Highlights

- Quickly eliminate many M_p candidates by finding small prime factors.
- Use the structure of possible factors of M_p to reduce wasted work.
- Show how often "trivial" factors appear in a range of exponents.

Goal

Explore how often Mersenne numbers $M_p = 2^p - 1$ (with prime p) have **small factors**, and how a lightweight sieve can filter candidates before running Lucas–Lehmer.

Background (quick refresher)

- *Mersenne numbers and primes refresher*

Research question

For prime exponents $p \leq P$ and a factor bound $q \leq Q$:

- how many M_p have a factor below Q?
- how much compute can be saved by sieving before Lucas–Lehmer?

Why this qualifies as a mathematical experiment

- **Finite procedure:** scan a finite range of p and candidate factors.
- **Observable(s):** first small factor found (if any), counts by p and by factor size.
- **Parameter space:** vary P and Q.
- **Outcome:** tables/plots that show factor frequency and sieve effectiveness.
- **Failure modes:** incomplete sieve bounds can mislead; clearly report Q as a limitation.

Experiment design

Computation

For fixed prime exponent p, any prime factor q of M_p satisfies:

- $q \equiv 1 \pmod{2p}$ (a common necessary condition used to generate candidates)

A practical sieve approach:

- Generate candidate primes q of the form $q = 2pk + 1$ up to Q.
- For each candidate prime q, check:
  - $2^p \equiv 1 \pmod{q}$

  If true, then q divides M_p.

Outputs

- table: per p, smallest factor found (or "none up to Q")
- plot: p vs. smallest factor (when present)
- summary: fraction of p eliminated by the sieve

How to run

```
make run EXP=e009
```

or:

```
uv run python -m mathxlab.experiments.e009
```

Notes / pitfalls

- This is a **bounded** sieve: "no factor found" only means "none found up to Q".
- Prime testing for q should be fast; for small Q, a simple deterministic method is fine.
- Report both P and Q prominently in the report.

Extensions

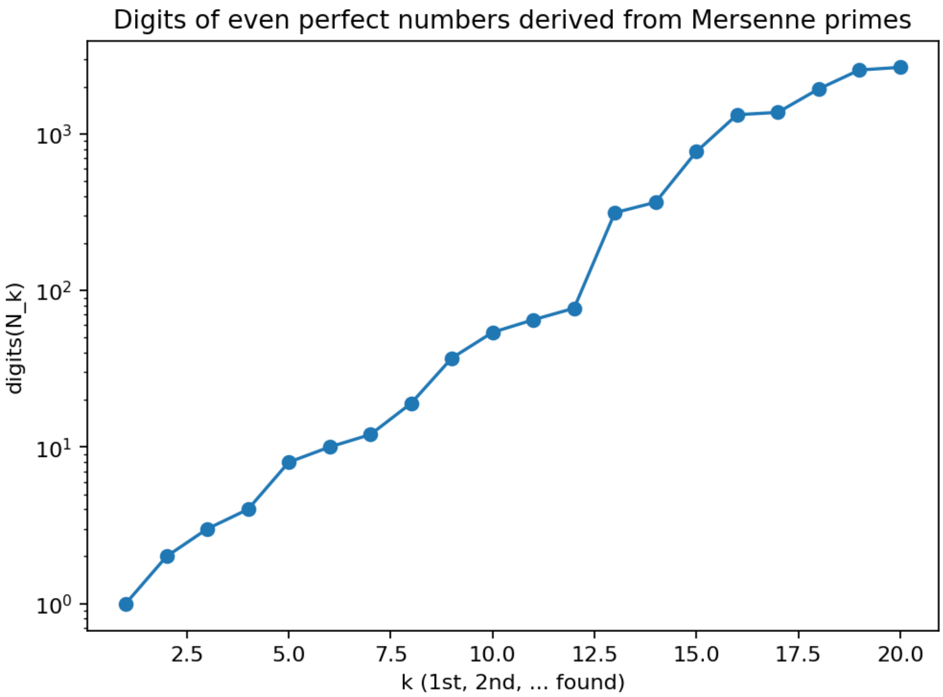
- Combine with E008: run Lucas–Lehmer only on exponents not eliminated by the sieve.
- Track how much wall time is saved by the combined pipeline.

References

See *References*.

[Cal21, con25b, Inc25b]

3.3.10 E010: Even perfect numbers from Mersenne primes



Tags: number-theory, quantitative-exploration, visualization (see *Valid Tags*)

Highlights

- Generate even perfect numbers via the Euclid–Euler theorem.
- Connect Mersenne prime exponents p to perfect numbers $N = 2^{p-1}(2^p - 1)$.
- Visualize growth and verify the defining property $\sigma(N) = 2N$ for sampled cases.

Goal

Make the Mersenne   perfect-number connection computationally explicit:

If $M_p = 2^p - 1$ is prime, then:

$$N_p = 2^{p-1}(2^p - 1)$$

is an **even perfect number**.

Background (quick refresher)

- *Mersenne numbers and primes refresher*
- *Perfect numbers refresher*

Research question

Across the Mersenne prime exponents discovered within your scan bounds:

- how fast do the corresponding even perfect numbers grow?
- can we verify perfectness ($\sigma(N) = 2N$) efficiently for these cases?

Why this qualifies as a mathematical experiment

- **Finite procedure:** find a set of Mersenne primes within a finite bound and generate perfect numbers.
- **Observable(s):** size metrics (digits), and validation checks of $\sigma(N) = 2N$.
- **Parameter space:** vary the exponent bound and validation depth.
- **Outcome:** concrete examples + growth plots that support intuition.
- **Reproducibility:** exponents tested and successes recorded in artifacts.

Experiment design

Computation

- Obtain a list of Mersenne prime exponents p from a scan (or a fixed list for small p).
- For each p, compute $N_p = 2^{p-1}(2^p - 1)$.
- Verify perfectness for these cases:

$$\sigma(N_p) = 2N_p.$$

For modest p, exact computation is feasible; for larger p, report size metrics and skip expensive checks.

Outputs

- table: p, M_p size, N_p size, and validation status
- plot: p vs digits of N_p
- optional: prime-factor structure display for small cases

How to run

```
make run EXP=e010
```

or:

```
uv run python -m mathxlab.experiments.e010
```

Notes / pitfalls

- Don't attempt divisor-sum sieves for huge N_p; validation must be bounded and explicit.
- Clearly separate "constructed from theorem" (conditional on M_p being prime) from "validated by computation".

Extensions

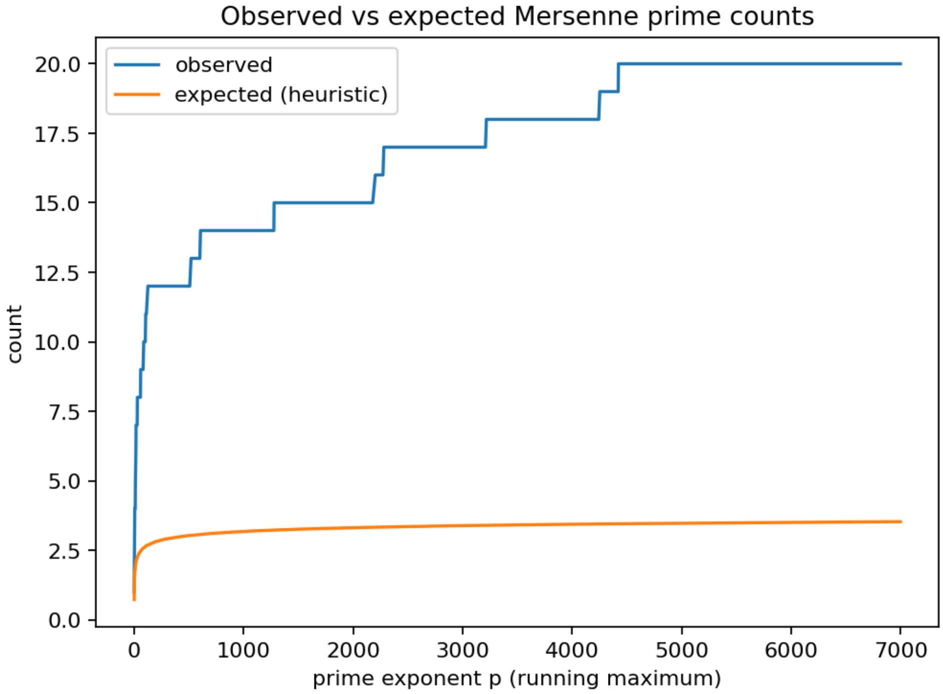
- Cross-link to E002 outputs for perfect numbers and compare growth on the same axes.
- Explore which parts of the perfectness check can be done symbolically using known factorization.

References

See *References*.

[Cald., con25b, Inc25a]

3.3.11 E011: Heuristic rarity of Mersenne primes



Tags: number-theory, quantitative-exploration, visualization (see *Valid Tags*)

Highlights

- Compare observed Mersenne-prime counts to a simple heuristic expectation curve.
- Visualize cumulative "found vs expected" as exponent bounds increase.
- Keep the discussion explicitly empirical (evidence, not proof).

Goal

Mersenne primes are extremely rare. This experiment compares:

- the observed count of Mersenne primes among prime exponents $p \leq P$
- a lightweight heuristic curve that grows slowly with P

The aim is intuition-building and trend visualization, not a rigorous model.

Background (quick refresher)

- *Mersenne numbers and primes refresher*

Research question

For increasing prime exponent bounds P:

- how does the cumulative count of "passes" from your scan grow?

- how does it compare to a simple expectation curve of the form:

$$E(P) \approx \sum_{p \le P,\ p\ \text{prime}} \frac{1}{p \ln 2}$$

Why this qualifies as a mathematical experiment

- **Finite procedure:** run a finite scan and compute a finite expected sum.
- **Observable(s):** cumulative count of passes vs. cumulative expected value.
- **Parameter space:** vary P (and optionally cap runtime per test).
- **Outcome:** plots that show agreement/divergence and suggest questions for deeper study.
- **Failure modes:** small ranges are noisy; the experiment should clearly label bounds.

Experiment design

Computation

- From E008 (or a fixed list), get the set of exponents tested and which passed.
- For each bound P, compute:
  - $\text{Found}(P)$ = number of passing exponents $p \le P$
  - $E(P)$ = cumulative heuristic sum
- Plot both curves against P.

Outputs

- plot: P vs Found(P) and E(P)
- table: selected checkpoints (P, Found, E)

How to run

```
make run EXP=e011
```

or:

```
uv run python -m mathxlab.experiments.e011
```

Notes / pitfalls

- The heuristic curve is a **comparison tool**, not a theorem.
- Results depend heavily on which exponents were actually tested; record the tested set in the report.
- A single large Mersenne prime can dominate impressions—keep axes and annotations clear.

Extensions

- Compare multiple heuristics (still empirical) and see which best matches the observed range.
- Extend the scan bound and see whether the deviation grows or stabilizes.

References

See *References*.

[Cal21, MR25]

3.4 Background

This section provides mathematical foundations for the experiments.

3.4.1 Mersenne numbers and primes refresher

This page is a *beginner-friendly* refresher for experiments about **Mersenne numbers** and **Mersenne primes**.

Core definitions

The **Mersenne numbers** are the integers

$$M_n = 2^n - 1 \quad (n \in \mathbb{N}).$$

A **Mersenne prime** is a Mersenne number that is prime:

$$M_p \text{ is a Mersenne prime} \iff 2^p - 1 \text{ is prime.}$$

A key (easy) fact is that if $2^n - 1$ is prime, then n must be prime.
So, in experiments, you typically only test M_p for *prime* exponents p. [con25b]

Key theorem / test: Lucas–Lehmer (why Mersenne primes are "computable")

For an odd prime exponent p, define $M_p = 2^p - 1$ and a sequence $\{s_k\}$ by

$$s_0 = 4, \qquad s_{k+1} = s_k^2 - 2.$$

Compute this sequence *modulo* M_p at each step, and then:

$$M_p \text{ is prime} \iff s_{p-2} \equiv 0 \pmod{M_p}.$$

This is the **Lucas–Lehmer test**. It's the workhorse behind most practical Mersenne-prime checks (and historically central to GIMPS). [con25a, MR24]

What experiments typically visualize

- **Growth with the exponent:** number of bits / digits of M_p as p grows.
- **Prime vs. composite behavior:** the Lucas–Lehmer residue $s_{p-2} \bmod M_p$ across many prime exponents.
- **Factor patterns for composites:** quickly finding small factors of M_p (to avoid running LLT when a trivial factor exists).
- **Connections to perfect numbers:** if M_p is prime, then $2^{p-1}(2^p - 1)$ is an even perfect number (Euclid–Euler). [Cald.]

For curated sequences (lists of exponents / known values), OEIS is a convenient "ground truth" reference. [Inc25a, Inc25b]

Practical numerical caveats

- **Always reduce modulo M_p in Lucas–Lehmer.**
  If you don't, the intermediate values explode in size (each squaring roughly doubles the bit-length).
- **Test the exponent first.**
  If p is composite, M_p is automatically composite, so there's no point running LLT.
- **Huge integers are fine, but you must be intentional.**
  Python's big integers won't overflow, but performance depends on bit-length and on the efficiency of modular squaring.
- **Separate "demo scale" from "real scale."**
  For educational experiments, keep p modest (e.g., $p \le 10^5$ is already huge for pure Python LLT). For large exponents, you'd rely on specialized implementations and careful FFT-based multiplication.

References

See *References*.

[Cal21, con25a, con25b, Inc25a, Inc25b]

3.4.2 Perfect numbers refresher

This page is a *beginner-friendly* refresher for experiments about **perfect numbers**. You only need basic number theory facts (divisors, primes) to follow it.

Core definitions

For a positive integer n, let

- $d \mid n$ mean "d divides n",
- $\sigma(n) = \sum_{d|n} d$ be the **sum-of-divisors function**,
- $s(n) = \sum_{d|n,\ d<n} d = \sigma(n) - n$ be the sum of **proper** divisors.

A number n is **perfect** iff its proper divisors sum to itself:

$$n \text{ is perfect} \iff s(n) = n \iff \sigma(n) = 2n.$$

Examples:

- 6 is perfect because $1 + 2 + 3 = 6$.
- 28 is perfect because $1 + 2 + 4 + 7 + 14 = 28$.

Key theorem: all even perfect numbers (Euclid–Euler)

A classic result completely characterizes **even** perfect numbers:

Euclid–Euler theorem.
An integer n is an even perfect number **iff**

$$n = 2^{p-1}(2^p - 1)$$

where $2^p - 1$ is prime (a **Mersenne prime**).

So every known perfect number is generated from a Mersenne prime exponent p. This is the main "generator" you'll use in experiments. [Cald., Voi98]

Why σ matters (multiplicativity)

If n factors as

$$n = \prod_{i=1}^{k} p_i^{a_i},$$

then

$$\sigma(n) = \prod_{i=1}^{k} \sigma\left(p_i^{a_i}\right) = \prod_{i=1}^{k} \frac{p_i^{a_i+1} - 1}{p_i - 1}.$$

This formula turns "sum all divisors" into a fast computation once you know the prime factorization.

The big open question: odd perfect numbers

It is **unknown** whether any **odd** perfect numbers exist. A large literature proves *constraints* (congruences, size bounds, number of prime factors, etc.). Experiments can explore these constraints (and why they make brute-force search unrealistic). [Guy04, OR14, Sto24]

What experiments typically visualize

Typical "lab" questions you can turn into plots and tables:

- **Verification by computation:** compute $\sigma(n)$ (via factorization) and check $\sigma(n) = 2n$ for candidates.
- **Generator experiment:** produce even perfect numbers from known Mersenne exponents p and confirm perfection.
- **Growth:** number of digits / bit length of $2^{p-1}(2^p - 1)$ as a function of p.
- **Divisor-function behavior:** compare $\sigma(n)/n$ (abundancy index) for random n vs. perfect numbers.
- **Odd constraints (toy models):** test necessary conditions on odd n and see how restrictive they are.

For data (lists of perfect numbers and exponents), OEIS is a convenient reference. [OEISFInc25]

Practical numerical caveats

Even with correct math, computation has a few traps:

- **Factorization dominates.** Computing $\sigma(n)$ via divisors is slow unless you factor n. For large n, factorization becomes infeasible; prefer the Euclid–Euler generator for even perfect numbers.
- **Big integers are fine, but expensive.** Python integers won't overflow, but operations on huge numbers scale with the number of bits (so be mindful in loops and plotting).
- **Prime testing vs. proof.** Testing that $2^p - 1$ is prime is nontrivial for large p. For experiments, use a curated list of known Mersenne prime exponents (e.g., from OEIS / GIMPS) rather than trying to discover new ones from scratch. [MersenneResearchIncGIMPS24, MersenneResearchIncGIMPS25]
- **Be explicit about definitions.** Some sources define "perfect" via $\sigma(n) = 2n$; others via proper divisors. Use one convention consistently in code and docs.

References

See *References*.

[Cald., Sto24, Voi98, OEISFInc25]

3.4.3 Taylor series refresher

This page is a *beginner-friendly* refresher for experiments that use Taylor polynomials. You only need basic calculus (derivatives) to follow it.

Taylor polynomial

Assume f has enough derivatives near x_0 (this is true for $\sin(x)$, $\cos(x)$, polynomials, exponentials, etc.). The Taylor polynomial of degree n around x_0 is

$$T_n(x; x_0) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!} (x - x_0)^k.$$

Intuition: $T_n(x; x_0)$ is the polynomial that matches $f(x_0)$ and the first n derivatives at x_0. It is usually accurate when x is close to x_0.

For $f(x) = \sin(x)$, the derivatives cycle, and around $x_0 = 0$ this becomes

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots .$$

Truncation error and the remainder

The approximation error is the remainder

$$R_n(x; x_0) = f(x) - T_n(x; x_0).$$

Under standard conditions, Taylor's theorem gives a remainder representation. One common form is the (Lagrange) remainder:

$$R_n(x; x_0) = \frac{f^{(n+1)}(\xi)}{(n+1)!} (x - x_0)^{n+1} \quad \text{for some } \xi \text{ between } x \text{ and } x_0.$$

This is the key qualitative message for experiments like E001:

- the factor $(x - x_0)^{n+1}$ makes the method **local** (good near x_0, potentially bad far away),
- increasing n helps most where $|x - x_0|$ is small.

What experiments typically visualize

In a numerical experiment, you often look at

- absolute error: $|R_n(x; x_0)|$
- relative error: $|R_n(x; x_0)|/|f(x)|$ (careful near zeros of f)

and plot them across a domain to see where the approximation is reliable.

Practical numerical caveats

Even when the mathematics are correct, computation can mislead:

- large $|x - x_0|$ and high n can produce huge intermediate terms,
- subtractive cancellation can reduce accuracy,
- floating-point rounding can dominate before the theoretical truncation error does.

A common "extension" experiment is to repeat the same plots using higher precision arithmetic to separate *truncation error* from *rounding error*.

Introductory reading

If you want a longer, *beginner-friendly* treatment (beyond this refresher), these are good starting points:

- A quick overview / definitions and examples: [Wikipediacontributors25d].
- A rigorous calculus textbook with a clean presentation of Taylor's theorem and remainders: [Apo91].
- A proof-oriented classic (slower, deeper): [Spi08].
- For the numerical viewpoint (truncation vs. rounding error): [BFB15].

References

See *References*.

[Apo91, BFB15, Spi08, Wikipediacontributors25d]

3.5 Getting started

This project uses an **uv-only** workflow and a small Makefile wrapper to run everything consistently.

3.5.1 Prerequisites

You need:

- **Python 3.14**
- **uv** on your PATH
- **GNU Make**

Windows notes

- Install GNU Make (e.g., via Chocolatey).
- You can run everything from `cmd.exe`, PowerShell, or from WSL.

3.5.2 Verify tools

From the repository root:

```
make uv-check
make python-check
make python-info
```

3.5.3 First-time setup

Create a virtual environment and install dependencies:

```
make venv
make install-dev
make install-docs
```

3.5.4 Run the full development chain

```
make final
```

`make final` runs:

- formatting (Ruff)
- linting (Ruff)
- type checking (mypy)
- tests (pytest)
- documentation (sphinx)

Documentation can be built separately via:

```
make docs
```

3.5.5 Run an experiment

Example (E001):

```
make run EXP=e001 ARGS="--seed 1"
```

See the experiment overview here: *Experiments Gallery*.

3.5.6 Build documentation locally

```
make docs
```

Output:

- `docs/_build/html`

3.5.7 Troubleshooting

`error: Failed to spawn: sphinx-build`

Sphinx is not installed in the uv environment.

Fix:

```
make install-docs
make docs
```

`make python-check` fails

The repo enforces Python **3.14**. Install Python 3.14, then recreate the environment:

```
make clean-venv
make venv
make install-dev
make install-docs
```

3.6 Development

This page describes the development workflow and the conventions used in this repository.

3.6.1 Workflow overview

Use the Makefile targets for everything. They wrap `uv run ...` so you do not need to activate a virtual environment manually.

Typical day-to-day:

```
make final
```

Documentation-only build:

```
make docs
```

Clean caches and build artifacts:

```
make clean
```

Remove the virtual environment (full reset):

```
make clean-venv
```

3.6.2 Makefile workflow

The Makefile is the **single entry point** for development tasks (env setup, quality checks, docs builds, and running experiments).

- Prefer `make final` before pushing.
- Prefer `make docs` to validate documentation changes.

- Use `make run EXP=<id>` to execute an experiment and write artifacts to `out/<id>/`.

Dependency groups

This summary is included from the Makefile documentation:

Dependencies are organized via `pyproject.toml` extras:

- **default**: runtime dependencies needed to run the package
- **dev** (`--extra dev`): developer tooling (ruff, mypy, pytest, etc.)
- **docs** (`--extra docs`): documentation tooling (sphinx, furo, myst-parser, sphinx-design, bibtex, …)

What the Makefile does

- Most dev targets run via:

```
uv run --extra dev ...
```

- Documentation targets run via:

```
uv run --extra docs ...
```

Why this matters

CI and local development can install only what they need:

- fast "dev chain" (no docs):

```
uv sync --extra dev
```

- docs build:

```
uv sync --extra docs
```

Run logs

Experiments live under `mathxlab/experiments/` and can be run either directly with Python or through Make targets (if available in your Makefile).

Run an experiment module directly

```
uv run python -m mathxlab.experiments.e001
```

Typical output locations (convention)

Depending on the experiment runner implementation, outputs are usually placed in:

- `out/e###/` (generated artifacts, figures, manifests)
- `docs/gallery/`, `docs/reports/`, `docs/manifests/` (published snapshots)

If you add new experiments, keep the numbering stable (`e001`, `e002`, …) so the gallery and documentation can remain consistent.

Common workflows

Task	Command
Complete quality check	`make final`
Build HTML docs	`make docs-html`
Run tests	`make test`
Clean all artifacts	`make clean`
Reset environment	`make clean && make venv`

Full reference

For the complete target-by-target reference and troubleshooting, see makefile.

3.6.3 Formatting, linting, typing, tests

- Formatting: **Ruff formatter**
- Linting: **Ruff**
- Typing: **mypy**
- Tests: **pytest**

CI formatting behavior

In CI, formatting runs in check mode (`ruff format --check`). Locally it formats in place.

3.6.4 Experiment authoring guidelines

When adding a new experiment:

1. Add a new module under `mathxlab/experiments/`, e.g. `e002_...py`.
2. Prefer deterministic outputs:
   - `--seed` argument if randomness is involved
   - write results to a single `--out` directory
3. Keep the experiment runnable as a module:
   - `python -m mathxlab.experiments.e002`
4. Update the docs:
   - add a short entry to *Experiments Gallery*
   - optionally add a dedicated page under `docs/experiments/` later

3.6.5 Documentation

Docs are built with Sphinx + MyST.

Build locally:

```
make install-docs
make docs
```

Deployed website:

- GitHub Pages from the `docs` workflow

3.6.6 Contributing (high-level)

- Create a feature branch.
- Open a PR against `main`.
- CI must pass before merge.
- Keep PRs small and well-scoped.

3.7 References

3.8 PDF download

A PDF build of this documentation is generated by GitHub Actions and published alongside the HTML site.

- Download PDF

If the link is missing, it usually means you are viewing an older deployment or the PDF build step failed.

BIBLIOGRAPHY

[Expa] Experimental mathematics (project euclid archive). URL: https://projecteuclid.org/journals/experimental-mathematics (visited on 2025-12-22).

[Expb] Experimental mathematics lab (university of luxembourg). URL: https://math.uni.lu/eml/ (visited on 2025-12-22).

[Expc] Experimental mathematics website. URL: https://www.experimentalmath.info (visited on 2025-12-22).

[Exp92] Experimental mathematics (journal). 1992. URL: https://www.tandfonline.com/journals/uexm20.

[Apo91] Tom M. Apostol. *Calculus, Volume 1*. John Wiley & Sons, 2 edition, 1991. ISBN 9780471000051. URL: https://books.google.com/books/about/Calculus\TU\textbackslash{}_Volume\TU\textbackslash{}_1.html?id=o2D4DwAAQBAJ.

[Arn15] Vladimir I. Arnold. *Experimental Mathematics*. MSRI Mathematical Circles Library. American Mathematical Society, 2015. ISBN 9780821894163. URL: https://bookstore.ams.org/msri-13/ (visited on 2025-12-22).

[BB05] David H. Bailey and Jonathan M. Borwein. Experimental mathematics: examples, methods and implications. *Notices of the American Mathematical Society*, 52(5):502–514, 2005. URL: https://www.ams.org/notices/200505/fea-borwein.pdf.

[BBC04] David H. Bailey, Jonathan M. Borwein, and Richard E. Crandall. Ten problems in experimental mathematics. *Experimental Mathematics*, 13(2):193–207, 2004.

[BvdPSZ14] Jonathan Borwein, Alf van der Poorten, Jeffrey Shallit, and Wadim Zudilin. *Neverending Fractions: An Introduction to Continued Fractions*. Volume 23 of Australian Mathematical Society Lecture Series. Cambridge University Press, 2014. ISBN 9780521186490. URL: https://www.cambridge.org/core/books/neverending-fractions/A3900DAB483D65CE6CB960A6B71226EE (visited on 2025-12-22).

[Bor05] Jonathan M. Borwein. The experimental mathematician: the pleasure of discovery and the role of proof. *International Journal of Computers for Mathematical Learning*, 10:75–108, 2005. doi:10.1007/s10758-005-6244-3.

[Bor09] Jonathan M. Borwein. *The Crucible: An Introduction to Experimental Mathematics*. A K Peters, Wellesley, MA, USA, 2009. ISBN 9781568813438.

[BB08] Jonathan M. Borwein and David H. Bailey. *Mathematics by Experiment: Plausible Reasoning in the 21st Century*. A K Peters, Wellesley, MA, USA, 2 edition, 2008. ISBN 9781568814421.

[BBG04] Jonathan M. Borwein, David H. Bailey, and Roland Girgensohn. *Experimentation in Mathematics: Computational Paths to Discovery*. A K Peters, Natick, MA, USA, 2004. ISBN 9781568811369. doi:10.1201/9781439864197.

[BBG+07] Jonathan M. Borwein, David H. Bailey, Roland Girgensohn, David Luke, and Victor H. Moll. *Experimental Mathematics in Action*. A K Peters, Wellesley, MA, USA, 2007. ISBN

9781568812717. URL: https://carmamaths.org/resources/jon/Preprints/Books/EMA/ema.pdf (visited on 2025-12-22).

[BB04] Jonathan M. Borwein and Richard P. Brent. *Inquiries into Experimental Mathematics*. A K Peters, Wellesley, MA, USA, 2004. ISBN 9781568812113.

[BFB15] Richard L. Burden, J. Douglas Faires, and Annette M. Burden. *Numerical Analysis*. Cengage Learning, 10 edition, 2015. ISBN 9781305253667. URL: https://www.cengage.com/c/numerical-analysis-10e-faires/9781305253667/.

[Cald.] Chris Caldwell. Characterizing all even perfect numbers. n.d. PrimePages (The Prime Database), proof note on the Euclid–Euler characterization. URL: https://t5k.org/notes/proofs/EvenPerfect.html (visited on 2025-12-25).

[Cal21] Chris K. Caldwell. Mersenne primes: history, theorems and lists. Online, 2021. PrimePages reference page collecting history, key theorems, and curated tables related to Mersenne primes and perfect numbers. URL: https://primes.utm.edu/mersenne/ (visited on 2025-12-27).

[con25a] Wikipedia contributors. Lucas–lehmer primality test — wikipedia, the free encyclopedia. Online, 2025. Revision as of 01:16, 31 October 2025. Abstract: Description and correctness of the Lucas–Lehmer test used to prove primality of Mersenne numbers \$M_p=2\^p-1\$. URL: https://en.wikipedia.org/w/index.php?oldid=1319643028&title=Lucas%E2%80%93Lehmer_primality_test (visited on 2025-12-27).

[con25b] Wikipedia contributors. Mersenne prime — wikipedia, the free encyclopedia. Online, 2025. Revision as of 20:10, 11 December 2025. Abstract: Overview of Mersenne primes (numbers of the form \$2\^p-1\$), their connection to perfect numbers, and known results and records. URL: https://en.wikipedia.org/w/index.php?oldid=1326946346&title=Mersenne_prime (visited on 2025-12-27).

[CP05] Richard Crandall and Carl Pomerance. *Prime Numbers: A Computational Perspective*. Springer, 2 edition, 2005. ISBN 9780387252827. Standard reference for computational prime testing and algorithms; includes discussion relevant to Mersenne primes and Lucas–Lehmer testing.

[Guy04] Richard K. Guy. *Unsolved Problems in Number Theory*. Springer, 3 edition, 2004. ISBN 9780387208602. URL: https://link.springer.com/book/10.1007/978-0-387-26677-0, doi:10.1007/978-0-387-26677-0.

[Inc25a] OEIS Foundation Inc. A000043 — mersenne exponents. Online, 2025. OEIS entry for primes \$p\$ such that \$2\^p-1\$ is prime (exponents of Mersenne primes). URL: https://oeis.org/A000043 (visited on 2025-12-27).

[Inc25b] OEIS Foundation Inc. A001348 — mersenne numbers: \$2\^p-1\$ where \$p\$ is prime. Online, 2025. OEIS entry for the Mersenne numbers sequence \$2\^p-1\$ with prime exponents \$p\$. URL: https://oeis.org/A001348 (visited on 2025-12-27).

[LCD+03] Shangzhi Li, Falai Chen, Jiansong Deng, Yaohua Wu, and Yunhua Zhang. *Mathematics Experiments*. World Scientific, 2003. ISBN 9789812380500. doi:10.1142/5008.

[MR24] Inc. Mersenne Research. Gimps — the math — primenet. Online, 2024. Background on the mathematics and algorithms used in the GIMPS search strategy (trial factoring, P-1, PRP testing, Lucas–Lehmer, double-checking). URL: https://www.mersenne.org/various/math.php (visited on 2025-12-27).

[MR25] Inc. Mersenne Research. List of known mersenne prime numbers. Online, 2025. PrimeNet/GIMPS list of known Mersenne primes including discovery metadata and verification method. URL: https://www.mersenne.org/primes/ (visited on 2025-12-27).

[OR14] Pascal Ochem and Michaël Rao. Lower bounds on odd perfect numbers. 2014. Slides (Montpellier, 2014-07-02). URL: https://www.lirmm.fr/~ochem/opn/opn\TU\textbackslash{}_slide.pdf (visited on 2025-12-25).

[PetkovvsekWZ96] Marko Petkovšek, Herbert S. Wilf, and Doron Zeilberger. *A=B*. A K Peters, Wellesley, MA, USA, 1996. ISBN 9781568810635. URL: https://sites.math.rutgers.edu/~zeilberg/expmath/ (visited on 2025-12-22).

[Polya54a] George Pólya. *Mathematics and Plausible Reasoning, Volume I: Induction and Analogy in Mathematics*. Princeton University Press, Princeton, NJ, USA, 1954.

[Polya54b] George Pólya. *Mathematics and Plausible Reasoning, Volume II: Patterns of Plausible Inference*. Princeton University Press, Princeton, NJ, USA, 1954.

[Spi08] Michael Spivak. *Calculus*. Publish or Perish, Inc., 4 edition, 2008. ISBN 9780914098911. URL: https://www.amazon.com/Calculus-4th-Michael-Spivak/dp/0914098918.

[Sto24] Andrew Stone. Improved upper bounds for odd perfect numbers — part i. *Integers: Electronic Journal of Combinatorial Number Theory*, 2024. URL: https://math.colgate.edu/~integers/y114/y114.pdf.

[Voi98] John Voight. Perfect numbers: an elementary introduction. 1998. Lecture notes / survey. Date: May 31, 1998; updated January 27, 2024. URL: https://jvoight.github.io/notes/perfelem-051015.pdf (visited on 2025-12-25).

[Wei03] Eric W. Weisstein. Perfect number. 2003. MathWorld—A Wolfram Web Resource. URL: https://mathworld.wolfram.com/PerfectNumber.html (visited on 2025-12-25).

[WK09] George Woltman and Scott Kurowski. On the discovery of the 45th and 46th known mersenne primes. *The Fibonacci Quarterly*, 46/47(3):194–197, 2009. Abstract PDF. Describes GIMPS methods and reports the discoveries of M37156667 and M43112609. URL: https://www.fq.math.ca/Abstracts/46_47-3/woltman.pdf (visited on 2025-12-27).

[MersenneResearchIncGIMPS24] Mersenne Research, Inc. (GIMPS). Mersenne prime discovery: 2^136279841-1 is prime! 2024. GIMPS press release page (52nd known Mersenne prime). URL: https://www.mersenne.org/primes/?press=M136279841 (visited on 2025-12-25).

[MersenneResearchIncGIMPS25] Mersenne Research, Inc. (GIMPS). Gimps milestones report. 2025. URL: https://www.mersenne.org/report\TU\textbackslash{}_milestones/ (visited on 2025-12-25).

[OEISFInc25] OEIS Foundation Inc. A000396: perfect numbers. 2025. The On-Line Encyclopedia of Integer Sequences (OEIS). URL: https://oeis.org/A000396 (visited on 2025-12-25).

[Wikipediacontributors25a] Wikipedia contributors. Harmonic number. Wikipedia, 2025. Permanent revision as of 20:03, 12 December 2025 (UTC). URL: https://en.wikipedia.org/w/index.php?oldid=1327129204\TU\textbackslash{}&title=Harmonic\TU\textbackslash{}_number (visited on 2025-12-25).

[Wikipediacontributors25b] Wikipedia contributors. Perfect number. Wikipedia, 2025. Permanent revision as of 13:46, 22 December 2025 (UTC). URL: https://en.wikipedia.org/w/index.php?oldid=1328905011\TU\textbackslash{}&title=Perfect\TU\textbackslash{}_number (visited on 2025-12-25).

[Wikipediacontributors25c] Wikipedia contributors. Prime number. Wikipedia, 2025. Permanent revision as of 20:47, 2 December 2025 (UTC). URL: https://en.wikipedia.org/w/index.php?oldid=1325385945\TU\textbackslash{}&title=Prime\TU\textbackslash{}_number (visited on 2025-12-25).

[Wikipediacontributors25d] Wikipedia contributors. Taylor series. Wikipedia, 2025. Permanent revision as of 04:10, 24 December 2025 (UTC). URL: https://en.wikipedia.org/w/index.php?oldid=1329166943\TU\textbackslash{}&title=Taylor\TU\textbackslash{}_series (visited on 2025-12-25).