catena

Architecture, Algorithm Decisions, and Formal Theory

v1.1.0-alpha

Lorenzo Silva Moore*

May 22, 2026

Contents

1 Introduction and Scope **3**
1.1 What continued fractions offer . . . 3
1.2 Scope . . . 3
1.3 Conventions . . . 4

2 Simple Continued Fractions[1] **5**
2.1 Core concepts and implementations . . . 5
2.1.1 Some useful properties of convergents . . . 9
2.2 Arithmetic and Algebraic Manipulations of SCFs . . . 12

3 Finite Simple Continued Fractions **15**
3.1 The Euclidean Algorithm as SCF Expansion[2] . . . 15
3.2 Properties and Operations of Finite SCFs . . . 17
3.3 Full Arithmetic on Finite SCFs . . . 18

4 Caching Strategy **21**
4.1 Motivation . . . 21
4.2 The `BaseCache` Primitive . . . 21
4.3 Generator-Level Caching: `CachedGenerator` . . . 22
4.4 Shared Caches and the Tail . . . 23

5 Periodic Simple Continued Fractions **24**
5.1 Purely periodic vs. eventually periodic . . . 24
5.2 The integer-part separation and its effect on the period . . . 24
5.3 Definition and data model . . . 24
5.4 Lagrange's theorem and quadratic surds . . . 25
5.4.1 Simplified form . . . 26
5.4.2 Normalized form . . . 27
5.4.3 Quadratic polynomials and the minimal polynomial . . . 27
5.4.4 Principal and conjugate surds . . . 29
5.4.5 Multiplicative inverse . . . 30
5.5 Conversion algorithms: quadratic surd ↔ PSCF . . . 32
5.5.1 Expanding a quadratic surd into a PSCF . . . 32

*Facultad de Ciencias (Faculty of Sciences), UNAM, México.
`lorenzosilvamoore@ciencias.unam.mx` ⊙ `lorenzo.smb.rayo@gmail.com`

5.5.2 Recovering a quadratic surd from a PSCF 36
5.6 Inherited and additional methods 37

6 Generators **39**
6.1 Structural operations 39
6.2 PeriodicGenerator 40

7 The `catena.numbers` module **42**
7.1 Constants 42
7.2 Random SCFs and the Gauss–Kuzmin distribution 42

8 Planned Extensions **45**
8.1 Arithmetic on Continued Fractions 45
8.2 Generalised Continued Fractions 45
8.3 Metric Theory and Gauss Map 45
8.4 Multidimensional Generalisations 45

9 Notation Index **46**

1 Introduction and Scope

This document records the mathematical foundations and algorithmic design decisions behind catena, a Python library for simple continued fractions. It is intended as a living document: new sections will be added as the library grows to cover further aspects of the theory.

The current implementation (`v0.4.0`) covers finite SCFs over the rationals, infinite generative SCFs, and periodic SCFs for quadratic irrationals.

1.1 What continued fractions offer

A real number can be represented in several ways: as a floating-point approximation, as an exact rational via `fractions.Fraction`, as a fixed-precision decimal, or as a continued fraction. Each representation makes different trade-offs.

Floats are fast and ubiquitous, but they introduce rounding error at every operation. A float is not a rational number; it is the nearest representable binary fraction, and the gap between the two is invisible unless you care to look for it.

`fractions.Fraction` is exact and supports arbitrary precision, but it accumulates large numerators and denominators quickly. Adding two fractions $p_1/q_1 + p_2/q_2$ yields $(p_1q_2 + p_2q_1)/(q_1q_2)$; in the worst case (coprime denominators) the denominator grows from n digits to $2n$ digits before any reduction, and every operation incurs a GCD computation whose cost grows with digit count.

Continued fractions offer a different kind of exactness. The partial quotients $[a_0;\, a_1, a_2, \ldots]$ are small integers — typically single digits for naturally occurring constants — so the representation is compact. The n-th convergent p_n/q_n is automatically in lowest terms by the determinant identity $p_nq_{n-1} - p_{n-1}q_n = \pm 1$, with no GCD step required. Convergents also have an optimality property: p_n/q_n is the best rational approximation to the true value among all fractions with denominator $\leq q_n$.

For irrational numbers, the continued fraction is the natural representation. $\sqrt{2} = [1; \overline{2}]$, $\varphi = [1; \overline{1}]$, and more generally every quadratic irrational has an eventually periodic expansion. Floats and decimals can only approximate; the SCF records the structure exactly.

These properties make SCFs useful for problems where rational approximation quality matters — gear-ratio design, modular arithmetic, Pell's equation, and the analysis of approximation algorithms, among others. catena is built around this perspective: the library works with the SCF itself as the primary object, deriving convergents and numeric approximations from it on demand.

1.2 Scope

catena provides three main abstractions:

- `SimpleContinuedFraction` — an infinite SCF $[a_0;\, a_1, a_2, \ldots]$ driven by an arbitrary callable generator. Convergents are memoised as they are requested; the cache is shared when the same tail is viewed with a different integer part (e.g. after an integer shift).
- `FiniteSimpleContinuedFraction` — a finite SCF $[a_0;\, a_1, \ldots, a_k]$ over $\mathbb{Q}$, constructible from a rational, float, or decimal string. Supports exact rational arithmetic and multiplicative inverse.
- `PeriodicSimpleContinuedFraction` — an eventually-periodic SCF $[a_0;\, a_1, \ldots, a_m, \overline{b_1, \ldots, b_r}]$ representing a quadratic irrational $(P + \sqrt{D})/Q$. Provides algebraic operations: conjugate, inverse, negation, and exact decimal evaluation.

Remark 1.1. `PeriodicSimpleContinuedFraction` is listed here for completeness. Its full mathematical treatment — Lagrange's theorem on quadratic irrationals, the Pell equation, and the algebraic operations (`conjugate`, `inverse`, `quadratic_coefficients`) — is deferred to a forthcoming section.

The mathematical content of this document is therefore structured around these three cases. It is not intended to be an exhaustive treatment of continued fraction theory; for that, see [3] and [5]. The goal is to give enough background to justify the algorithms and design choices in the library.

1.3 Conventions

Throughout this document:

- $\mathbb{N}$ denotes the natural numbers (including zero).
- For any $n \in \mathbb{N}$, we implicitly assume the set-theoretic construction $n = \{0, 1, \ldots, n-1\}$.
- $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ denote the integers, rationals, and reals respectively.
- Likewise, $\mathbb{Z}^+$, $\mathbb{Q}^+$, $\mathbb{R}^+$ denote the positive (non-zero) integers, rationals, and reals respectively.
- $[a_0;\, a_1, a_2, \ldots, a_n]$ denotes a finite SCF with $a_0 \in \mathbb{Z}$ and $a_k \in \mathbb{Z}^+$ for $k \geq 1$.
- $c_k = (p_k, q_k)$ denotes the k-th convergent numerator–denominator pair, with $q_k \in \mathbb{Z}^+$ by convention.
- The function $\{\cdot\} : \mathbb{R} \to \mathbb{R}$ denotes the fractional part, so $\{x\} = x - \lfloor x \rfloor$.

Remark 1.2. We follow the set-theoretic convention $\mathbb{N} = \{0, 1, 2, \ldots\}$ including zero. The positive integers $\mathbb{Z}^+ = \{1, 2, 3, \ldots\}$ exclude zero. In particular, all body partial quotients a_k (for $k \geq 1$) lie in $\mathbb{Z}^+$, while the integer part $a_0 \in \mathbb{Z}$ may be zero or negative.

2 Simple Continued Fractions[3]

In the most general sense, a continued fraction is an expression of the form

$$a_0 + \cfrac{b_1}{a_1 + \cfrac{b_2}{a_2 + \cfrac{b_3}{\ddots}}}$$

where $a_0, a_1, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$, depending on the context, may be assumed to be integers, rational numbers, either real numbers or complex numbers, and even functions. For the purposes of catena, and by extension this document, we we restrict attention to the so-called *simple continued fractions* (SCFs), that is, those where $b_n = 1$ for all n and a_n are integers with $a_1, a_2, \ldots, a_n, \ldots > 0$.[4] Namely, we are left with expressions of the form

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots}}}$$

with $a_0 \in \mathbb{Z}$ and $a_1, a_2, \ldots, a_n, \ldots \in \mathbb{Z}^+$. For brevity, it is common to write this expression as $[a_0;\, a_1, a_2, \ldots, a_n, \ldots]$. The a_n are called the *partial quotients* of the SCF, and the n-th *convergent* is the rational number obtained by truncating the expansion at a_n, that is, $[a_0;\, a_1, a_2, \ldots, a_n]$.

As any sequence of indexable objects, one could represent an SCF as a function $f : \mathbb{N} \to \mathbb{Z}$ such that $f(0) \in \mathbb{Z}$ and $f(n) \in \mathbb{Z}^+$ for $n > 0$, thereby encoding the entire expansion uniformly within a single map. However, as suggested by the standard notation $[a_0;\, a_1, a_2, \ldots]$, it is often conceptually and computationally advantageous to distinguish and separate the integer part from the remaining partial quotients.

Accordingly, in this work we adopt a split representation: we treat the first partial quotient $a_0 \in \mathbb{Z}$ as the *integer part* of the SCF, and encode the remaining coefficients via a generator $f : \mathbb{N} \to \mathbb{Z}^+$ where $f(n) = a_{n+1}$. This distinction is not merely notational; it reflects how SCFs are constructed and manipulated algorithmically in catena, and will play a central role in the implementations developed in the following sections.

2.1 Core concepts and implementations

Definition 2.1 (Generating Function)**.** Let $I \in \mathbb{N} \cup \{\mathbb{N}\}$. A *generator* indexed by I is any function $g : I \to \mathbb{Z}^+$.

When $I = \mathbb{N}$, we say that g is an *infinite generator*, and when $I \in \mathbb{N}$, we say that g is a *finite generator*. Furthermore, in the finite case, we can say that g is a I*th-order* generator.

Remark 2.2. Note we allow $I = 0$. In this case, the generator is vacuous and the corresponding SCF reduces to $[a_0]$ for some $a_0 \in \mathbb{Z}$.

catena — `Generator`, `CachedGenerator`, `FiniteGenerator`

```
from catena.generators import Generator, CachedGenerator,
```

[3] As with all the other sections, the following is not meant to be an exhaustive treatment of the topic, but rather a sketch of the main results and algorithms related to SCFs that are relevant to catena. For a more comprehensive treatment, see [3] and [5].

[4] In the literature is common to allow $a_0 \in \mathbb{R}$ (for instance, see [3]), but for the sake of avoiding floating-point arithmetic, as well as to gain a notion of uniqueness, we will restrict it to $\mathbb{Z}$. If nothing more, we do not lose much generality since any real number admits a unique SCF expansion with integer part $a_0 = \lfloor x \rfloor$ and fractional part $x - a_0 = 1/(a_1 + 1/(a_2 + \ldots))$. While we won't stop to prove it here, it is worth noting that the SCF expansion of a rational number is finite, while that of an irrational number is infinite.

```
    FiniteGenerator

# Infinite generator: every partial quotient equals 2 (body of sqrt
    (2) = [1; 2, 2, 2, ...])
gen = Generator(lambda n: 2)

# CachedGenerator is appropriate when g(n) is expensive and will be
    queried
# repeatedly, or when the same generator is shared across several
    SCF
# instances.  It memoises each top-level call in an OrdinalCache and
     exposes
# catena's cache introspection layer (call_count, read_count,
    reset_cache).
# Example: g(n) = number of divisors of (n+2) that do not exceed
    sqrt(n+2)
import math
def small_divisors(n):
    m = n + 2
    return sum(1 for d in range(1, int(math.isqrt(m)) + 1) if m % d
        == 0)

cgen = CachedGenerator(small_divisors)
cgen(100)          # computes small_divisors(100), stores result
cgen(100)          # O(1) cache hit
cgen.call_count    # 1  (one miss)
cgen.read_count    # 1  (one hit)
# See the Caching section for the full cache infrastructure.

# Finite generator: body partial quotients of pi = [3; 7, 15, 1,
    292, ...]
# The integer part 3 is stored separately in the SCF, not in the
    generator.
fgen = FiniteGenerator([7, 15, 1, 292, 1, 1, 1, 2])  # a_1 ... a_8
fgen(0)   # 7   (a_1)
fgen(3)   # 292 (a_4)
fgen(10)  # raises IndexError (only indices 0--7 are defined)
```

Definition 2.3 (Simple Continued Fraction)**.** A *simple continued fraction* is an expression of the form

$$x = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots}}}$$

where a_0 is an integer and $a_1, a_2, \ldots, a_n, \ldots$ are positive integers. Thus, given $a_0 \in \mathbb{Z}$ and a generator $g : I \to \mathbb{Z}^+$, such that $g(n) = a_{n+1}$, we represent the SCF as the ordered pair $x = (a_0, g)$.

catena — SimpleContinuedFraction, FiniteSimpleContinuedFraction

```
from catena import SimpleContinuedFraction,
    FiniteSimpleContinuedFraction

# Infinite SCF: golden ratio phi = [1; 1, 1, 1, ...]
phi = SimpleContinuedFraction(lambda n: 1, integer_part=1)

# Finite SCF: 355/113 = [3; 7, 16]
```

```
scf = FiniteSimpleContinuedFraction.from_rational((355, 113))
scf.integer_part # 3
scf.generator    # FiniteGenerator([7, 16])
```

Definition 2.4 (Segment)**.** Let $x = (a_0, g)$ be a simple continued fraction, where $g : I \to \mathbb{Z}^+$ is a generator. We define the *segment* of x at index $n \in I$ as the SCF obtained by truncating the first n partial quotients from g.

Formally, for $n \in I$, the segment of x at index n is given by $x_n = (a_0, g_n)$, where $g_n := g|_n$ is the restriction of g to n, i.e. the function with domain $\{0, 1, \ldots, n-1\}$ satisfying $g_n(k) = g(k)$.

Remark 2.5. In particular, $x_0 = (a_0, \varnothing)$ is the degenerate finite SCF with no body partial quotients, representing the integer a_0 itself. For $n \geq 1$, $x_n = [a_0;\, g(0), g(1), \ldots, g(n-1)]$.

catena — SimpleContinuedFraction.segment

```
from catena import SimpleContinuedFraction

# Example: for phi = [1; 1, 1, 1, ...], the segment at index n is
    also phi
phi = SimpleContinuedFraction(lambda n: 1, integer_part=1)
phi.segment(0) # [1;]
phi.segment(1) # [1; 1]
phi.segment(10) # [1; 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Remark 2.6. Given that the segment is defined for $n \in I$, for finite SCFs, the segment is only defined for $n \leq$ size. For $n >$ size, the segment is not defined since it would require access to partial quotients that do not exist in the finite generator. Therefore, in the finite case, calling `segment(n)` with $n >$ size will raise an `IndexError`.

As for `segment(size)`, `self.shift(0)` is returned since the segment at index `size` corresponds to the original SCF itself. `shift(0)` creates a new `FiniteSimpleContinuedFraction` that shares the same `FiniteGenerator` and convergent cache as `self`, but with `integer_part` unchanged (shifting by zero). This avoids a deep copy while preserving the immutability contract.

Definition 2.7 (Remainder)**.** Let $x = (a_0, g)$ be a simple continued fraction, where $g : I \to \mathbb{Z}^+$. For $n \in I$, we define the *remainder* of x at index n as the SCF

$$r_n(x) = (g(n), h),$$

where $h : J \to \mathbb{Z}^+$, with J the ordinal $|I \setminus \{0, 1, \ldots, n\}|$, is defined by $h(m) = g(m + n + 1)$. Concretely, J is identified with $I \setminus \{0, 1, \ldots, n\}$ via the shift $m \mapsto m + n + 1$.

Remark 2.8. As of now, catena does not have a dedicated implementation for remainders. We mention them here because they will come up in some of the theoretical results.

Definition 2.9 (Convergent)**.** Let $x = (a_0, g)$ be a simple continued fraction, where $g : I \to \mathbb{Z}^+$. For $n \in I$, we define the n-th *convergent* of x as the rational number

$$c_n = \frac{p_n}{q_n}$$

given by the value of the segment x_n.

Remark 2.10 (Segment vs Convergent)**.** The segment x_n is itself a (finite) SCF, while the convergent p_n/q_n is a rational number. They are related by evaluation: x_n represents p_n/q_n, and every rational number admits a unique finite SCF expansion of this form.

Definition 2.11 (Tail)**.** Let $x = (a_0, g)$ be a simple continued fraction, where $g : I \to \mathbb{Z}^+$. For $n \in I$, the *tail* of x is the SCF given by $t(x) = (0, g)$. This is to say, the tail of x is the SCF corresponding to the fractional part of x, $\{x\}$.

Likewise, the n-*th tail convergent* of x is the n-th convergent of $t(x)$.

```
catena — SimpleContinuedFraction.tail
from catena import SimpleContinuedFraction

# Example: for phi = [1; 1, 1, 1, ...], the tail is also phi
phi = SimpleContinuedFraction(lambda n: 1, integer_part=1)
phi.tail() # [0; 1, 1, 1, ...]

phi.tail_convergent(7) # 21/34
phi.tail().convergent(7) # 21/34
```

Theorem 2.12 (Recurrence Relations for Tail Convergents[5])**.** *Let $x = (a_0, g)$ be a simple continued fraction, where $g : I \to \mathbb{Z}^+$. For $n \in I$, define the n-th tail convergent by*

$$\frac{h_n}{k_n} = [0;\, g(0), g(1), \ldots, g(n)].$$

Then the numerators and denominators satisfy the recurrence relations

$$h_n = g(n)\, h_{n-1} + h_{n-2}, \quad k_n = g(n)\, k_{n-1} + k_{n-2} \qquad (n \geq 0),$$

with initial conditions

$$h_{-2} = 1, \quad h_{-1} = 0, \qquad k_{-2} = 0, \quad k_{-1} = 1.$$

Remark 2.13. We extend the indexing of the sequences (h_n) and (k_n) to include the values $n = -2$ and $n = -1$, with

$$h_{-2} = 1, \quad h_{-1} = 0, \qquad k_{-2} = 0, \quad k_{-1} = 1.$$

These values should be understood as formal initial conditions for the recurrence, rather than as convergents. This convention allows the recurrence relations to hold uniformly for all $n \geq 0$ without requiring special cases, which is convenient for implementation.

Corollary 2.14 (Convergent from Tail Convergents)**.** *Let $x = (a_0, g)$ be a simple continued fraction, and let $\frac{h_n}{k_n} = [0;\, g(0), \ldots, g(n)]$ be the n-th tail convergent. Then the n-th convergent of x is given by*

$$c_n = \frac{p_n}{q_n} = a_0 + \frac{h_n}{k_n} = \frac{a_0 k_n + h_n}{k_n}.$$

Remark 2.15 (Caching Tail Convergents)**.** The recursive nature of the tail convergents makes them ideal candidates for caching. By storing previously computed tail convergents, we can avoid redundant calculations and significantly improve the efficiency of algorithms that rely on them. A pretty direct example would be the scenario of iterating over the convergents of a given SCF: by caching the tail convergents, we can compute each new convergent in constant time after an initial linear time setup, rather than having to recompute the entire tail convergent from scratch for each new convergent.

```
catena — SimpleContinuedFraction.tail_convergent
    from catena import SimpleContinuedFraction

    # All SCFs have a dedicated cache for tail convergents.
    phi = SimpleContinuedFraction(lambda n: 1, integer_part=1)

    phi.cache_handler # CacheHandler for tail convergents; owns the
```

[5]See, for instance, [3], Theorem 1, p. 4. For a more general version of this result, see [2], Theorem 2.1, p. 22.

```
    cache and handles caching logic.
phi.tail_convergent(0) # computes and caches the 0-th tail
    convergent (1/1)
phi.tail_convergent(1) # computes and caches the 1-st tail
    convergent (1/2), reusing the 0-th tail convergent from cache

tail = phi.tail() # the tail is also phi, so it shares the same
    cache of tail convergents
tail.cache_handler is phi.cache_handler # True
tail.tail_convergent(1) # retrieves the 1-st tail convergent
    from cache (1/2)
```

Remark 2.16 (Shared `tail_convergent` Callable)**.** `tail()` uses `_from_shared`, which copies the reference to `_cache_handler` *and* to the `SetLightCache`-wrapped `tail_convergent` callable. Since the cache lives in `cache_handler`, both objects read from and write to the same `OrdinalCache` instance, giving true shared memoisation. This is why `tail.tail_convergent(1)` above retrieves a cached value rather than recomputing it.

In the literature, tail convergents are typically not introduced explicitly, as they are not of primary theoretical interest. However, they provide a useful computational tool. In particular, they allow distinct SCFs to share both the same generator and the same cache of previously computed values, thereby avoiding redundant computations and improving efficiency.

Another point of divergence from the standard presentation is the indexing of convergents. In the literature, the n-th convergent is often defined as $[a_0;\, a_1, \ldots, a_n]$, which corresponds to our $(n-1)$-th convergent. We instead adopt zero-based indexing to reflect the conventions of Python, so that the first tail convergent corresponds to

$$[0; g(0)] = \frac{1}{g(0)} = \frac{1}{a_1}.$$

While minor, this distinction is worth keeping in mind when comparing results with the classical literature.

Remark 2.17 (RecursionError for Negative Indices)**.** The recurrence relations for tail convergents are only defined for $n \geq 0$, and the seed values are defined for $n = -1$ and $n = -2$. Therefore, if the recurrence is called with $n < -2$, it indicates a logic error in the implementation, as it would require access to tail convergents that are not defined.

In `catena`, this condition raises a `RecursionError`. Python's `RecursionError` is conventionally associated with call-stack exhaustion, but here it is repurposed to signal that the recurrence has descended below its valid base cases — a situation that can only arise from a logic error upstream, not from a bad user-supplied argument. An alternative design would use `ValueError`; the current choice is a deliberate signal that the error is structural (a malformed recursive call) rather than an invalid input.

2.1.1 Some useful properties of convergents

Theorem 2.18 (Determinant Identity for Convergents[6])**.** *Let $x = (a_0, g)$ be a simple continued fraction, where $g : I \to \mathbb{Z}^+$. For $n \in I$, let $\frac{p_n}{q_n}$ be the n-th convergent of x. Then the following identity holds:*

$$p_n q_{n-1} - p_{n-1} q_n = (-1)^{n+1} \qquad (n \in I).$$

Corollary 2.19 (Coprimality of Convergents)**.** *Let $x = (a_0, g)$ be a simple continued fraction, where $g : I \to \mathbb{Z}^+$. For $n \in I$, let $\frac{p_n}{q_n}$ be the n-th convergent of x. Then p_n and q_n are coprime for all $n \in I$; that is,*

$$\gcd(p_n, q_n) = 1 \qquad (n \in I).$$

[6] See, for instance, [3], Theorem 1, p. 4.

Proof. From the determinant identity,

$$p_n q_{n-1} - p_{n-1} q_n = (-1)^{n+1},$$

any common divisor of p_n and q_n must divide the left-hand side, and hence divide $(-1)^{n+1}$. Therefore, $\gcd(p_n, q_n) = 1$. □

Remark 2.20. In particular, each convergent is already expressed in lowest terms. This justifies the design choice of not delegating convergent arithmetic to the `Fraction` class from the Python standard library, which would redundantly attempt to reduce fractions.

Since the recurrence relations preserve coprimality automatically, it is more efficient to maintain numerators and denominators directly, avoiding both the overhead of normalization and the cost of repeatedly instantiating `Fraction` objects. Many other algorithms in `catena` inherit this property as well, allowing coprimality to be preserved without additional (and redundant) computation.

Corollary 2.21 (Difference of Consecutive Convergents)**.** *Let $x = (a_0, g)$ be a simple continued fraction, where $g : I \to \mathbb{Z}^+$. For $n \in I$, let $\frac{p_n}{q_n}$ be the n-th convergent of x. Then the following identity holds:*

$$\frac{p_n}{q_n} - \frac{p_{n-1}}{q_{n-1}} = \frac{(-1)^{n+1}}{q_n q_{n-1}} \qquad (n \in I).$$

Theorem 2.22. *Let $x = (a_0, g)$ be a simple continued fraction, where $g : I \to \mathbb{Z}^+$. For $n \in I$, let $\frac{p_n}{q_n}$ be the n-th convergent of x. Then the following identity holds:*

$$p_n q_{n-2} - p_{n-2} q_n = (-1)^n g(n) \qquad (n \in I, n \geq 1).$$

Corollary 2.23. *Let $x = (a_0, g)$ be a simple continued fraction, where $g : I \to \mathbb{Z}^+$. For $n \in I$, let $\frac{p_n}{q_n}$ be the n-th convergent of x. Then the following identity holds:*

$$\frac{p_n}{q_n} - \frac{p_{n-2}}{q_{n-2}} = \frac{(-1)^n g(n)}{q_n q_{n-2}} \qquad (n \in I, n \geq 1).$$

Theorem 2.24 (Monotonicity and Convergence of Convergents)**.** *Let $x = (a_0, g)$ be a simple continued fraction, where $g : I \to \mathbb{Z}^+$. For $n \in I$, let $\frac{p_n}{q_n}$ be the n-th convergent of x. Then the following properties hold:*

1. *The sequence of convergents alternates around x: for even n, we have $c_n > x$, while for odd n, we have $c_n < x$.*[7]
2. *The sequence of convergents is strictly decreasing for even indices and strictly increasing for odd indices.*
3. *The sequence of convergents converges to x as $n \to \infty$ (if $I = \mathbb{N}$).*

Remark 2.25. The monotonicity and convergence properties of convergents are fundamental to their role as approximations of the original SCF. The alternating nature of the sequence ensures that each new convergent provides a better approximation than the previous one, while the convergence guarantees that we can get arbitrarily close to x by taking sufficiently many convergents.

This is, among other things, the theoretical basis for being able to define a generic `__float__` method for SCFs, which is implemented in `catena` as the value of a fixed convergent call. Namely, `.convergent(50)` is used as the default approximation for the float value of an arbitrary

[7]Due to our indexing convention (where $c_0 = [a_0;\, a_1]$), the parity of the alternating inequalities is reversed relative to some standard references.

SCF. The `machine_epsilon` constant from the Python standard library is usually around $2^{-52} \approx 2.22 \times 10^{-16}$, so the 50-th convergent of any SCF will be accurate to well within machine precision. A heuristic train of thought for this choice is as follows: since the sequence of convergents alternates around x and converges to x, we focus on the value of the difference between the n-th convergent and $(n-1)$-th convergent, which, per Corollary 2.21, is given by

$$\left|\frac{p_n}{q_n} - \frac{p_{n-1}}{q_{n-1}}\right| = \frac{1}{q_n q_{n-1}}.$$

On the other hand, the slowest growth rate for the denominators of convergents is given by the Fibonacci sequence, which corresponds to the case where $g(n) = 1$ for all n. In this case, we can compute that

$$\frac{1}{q_{50}q_{49}} = \frac{1}{20365011074 \cdot 32951280099} = \frac{1}{671053184118610816326} \approx 1.49 \times 10^{-21}.$$

Therefore, even in the worst-case scenario, the 50-th convergent will be accurate to well within machine precision.

catena — `SimpleContinuedFraction.__float__`

```
from catena import SimpleContinuedFraction

# Example: for phi = [1; 1, 1, 1, ...], the float value is
    approximately 1.618033988749895
phi = SimpleContinuedFraction(lambda n: 1, integer_part=1)

# We can call float() on phi directly.
float(phi) # approximately 1.618033988749895
```

Remark 2.26 (No `to_decimal` for Infinite SCFs)**.** The finite SCF class provides a `to_decimal` method that returns an exact arbitrary-precision `Decimal` value (see §3). No analogous method is defined on `SimpleContinuedFraction`.

The reason is that `to_decimal` requires choosing *which* convergent to evaluate, and for a finite SCF there is a canonical choice: the terminal convergent, which is the exact rational value. For an infinite SCF no such canonical convergent exists — the sequence of convergents converges to x, but every individual convergent is only an approximation. Any fixed index n would give an answer whose precision depends on n and on the specific SCF, with no guarantee of meeting a caller-supplied precision target.

By contrast, `__float__` commits to IEEE 754 double precision ($\approx 2^{-52}$), which is fixed by the hardware; convergent (50) is always sufficient (see the remark above). For truly arbitrary-precision output the caller should use `segment(`n`).to_decimal()` for a chosen n, making the precision trade-off explicit.

Theorem 2.27 (Evaluation from Remainder[8])**.** *Let $x = (a_0, g)$ be a simple continued fraction, where $g : I \to \mathbb{Z}^+$. For $n \in I$, let $\frac{p_n}{q_n}$ be the n-th convergent of x, and let r_n be the n-th remainder of x. Then the following identity holds:*

$$x = \frac{p_{n-1}r_n + p_{n-2}}{q_{n-1}r_n + q_{n-2}} \qquad (n \in I).$$

Remark 2.28. In this theorem r_n denotes the *value* of the n-th remainder SCF $r_n(x) = (g(n), h)$ as a real number, not the SCF object itself. The identity then follows from the matrix formulation of the recurrence (see, e.g., [3], Theorem 5).

[8] See [3], Theorem 5, p. 7 & Theorem 7, p. 8.

2.2 Arithmetic and Algebraic Manipulations of SCFs

In full generality, arithmetic operations on SCFs are nontrivial, since the generators of the resulting SCFs may not admit closed-form expressions. However, there are some special cases where we can derive explicit formulas for the generators of the resulting SCFs, allowing us to perform certain operations efficiently. We present two such cases: the multiplicative inverse and the additive inverse.[9]

Proposition 2.29 (SCF of the Multiplicative Inverse)**.** *Let $x = (a_0, g)$ be an SCF with $x > 0$.*

1. *If $a_0 = 0$: $x = [0;\, a_1, a_2, \ldots]$ with $a_1 \geq 1$, so $1/x = [a_1;\, a_2, a_3, \ldots]$.*
2. *If $a_0 > 0$: $x = [a_0;\, a_1, a_2, \ldots]$ with $a_0 \in \mathbb{Z}^+$, so $1/x = [0;\, a_0, a_1, a_2, \ldots]$.*

In both cases the partial quotients of $1/x$ lie in $\mathbb{Z}^+$: in case 1, the new integer part is $a_1 \geq 1$ and the body continues with positive values; in case 2, the body begins with $a_0 \geq 1$.

The condition $x > 0$ (equivalently $a_0 \geq 0$) is necessary. If $a_0 < 0$ then $x < 0$, so $1/x < 0$ and its SCF has a negative integer part; this cannot be recovered by the simple generator-prepend transformation above, which would place a negative value in a body slot (violating $g : I \to \mathbb{Z}^+$). The result is computed by `SimpleContinuedFraction.inverse`, which stores it in a write-once `_inverse` attribute and sets the inverse's own `_inverse` back to `self`, making `inverse` an involution on cached pairs.

Remark 2.30. Reciprocity of the inverse operation is a consequence of the fact that the SCF of $1/x$ is constructed by a simple transformation of the generator of x, and this transformation is designed to be reversible. The caching mechanism ensures that once the inverse is computed, it can be retrieved in constant time on subsequent calls, and that the involution property holds without redundant computation.

catena — SimpleContinuedFraction.inverse

```
phi = SimpleContinuedFraction(lambda n: 1, integer_part=1)
# phi = [1; 1, 1, 1, ...] (golden ratio, a_0 = 1 > 0)
inv_phi = phi.inverse()    # 1/phi = phi - 1 = [0; 1, 1, 1, ...]
inv_phi.integer_part       # 0
inv_phi.inverse() is phi   # True  (involution: write-once _inverse
   points back to self)

# a_0 = 0 case:
tail_phi = phi.tail()           # [0; 1, 1, 1, ...]
tail_phi.inverse().integer_part  # 1  (= a_1; inverse is phi itself)
```

Remark 2.31. In Proposition 2.29, we saw that the SCF of the multiplicative inverse of a positive SCF can be obtained almost for free. By contrast, the negative case is more involved, and there is no simple generator transformation that yields the SCF of $1/x$ when $x < 0$. However, if we could define the additive inverse of an SCF ($-x$ for any x), then we could compute the multiplicative inverse of a negative SCF by first negating it and then applying the positive case. In short, we would have $1/x = -(1/(-x))$, and since $-x > 0$, we could apply the positive case to compute $1/(-x)$, and then negate the result to get $1/x$.

Proposition 2.32 (SCF of the Additive Inverse)**.** *Let $x = (a_0, g)$ be an SCF with $g \neq \varnothing$, and let $a_1 := g(0)$. Then*

$$-x = (-a_0 - 1,\ g^*),$$

[9] There is a way to define basic arithmetic for SCFs, proposed by Gosper [1], but the implementation is a tad more cumbersome and there are also some caveats. Therefore, implementing the Gosper arithmetic for continued fractions is deferred to a later version of catena.

where $g^*\colon J \to \mathbb{Z}^+$ *is the generator defined by*

$$g^* = \begin{cases} (1,\ a_1 - 1,\ g(1),\ g(2),\ \ldots) & \text{if } a_1 > 1, \\ (g(1) + 1,\ g(2),\ g(3),\ \ldots) & \text{if } a_1 = 1. \end{cases}$$

Proof. Write $x = a_0 + \alpha$, where $\alpha := [0;\, g(0), g(1), \ldots] \in (0, 1)$ is the fractional part of x. Then

$$-x \;=\; -a_0 - \alpha \;=\; \underbrace{(-a_0 - 1)}_{\in\mathbb{Z}} + (1 - \alpha),$$

so the integer part of $-x$ is $-a_0 - 1$ and its fractional part is $1 - \alpha \in (0, 1)$. It remains to find the SCF expansion of $1 - \alpha$.

Setting $\beta := [0;\, g(1), g(2), \ldots]$, we have $\alpha = \dfrac{1}{a_1 + \beta}$, so

$$1 - \alpha = \frac{a_1 - 1 + \beta}{a_1 + \beta}, \qquad \frac{1}{1 - \alpha} = 1 + \frac{1}{(a_1 - 1) + \beta}.$$

Case $a_1 > 1$**.** Here $a_1 - 1 \geq 1$, so $(a_1 - 1) + \beta$ is a positive real number with integer part $a_1 - 1$ and fractional part β. Thus

$$(a_1 - 1) + \beta = [a_1 - 1;\ g(1),\ g(2), \ldots],$$

and therefore

$$\frac{1}{1 - \alpha} = 1 + \frac{1}{[a_1 - 1;\ g(1),\ g(2), \ldots]} = [1;\ a_1 - 1,\ g(1),\ g(2), \ldots].$$

Hence $1 - \alpha = [0;\ 1,\ a_1 - 1,\ g(1),\ g(2), \ldots]$.

Case $a_1 = 1$**.** Here $a_1 - 1 = 0$, so

$$\frac{1}{1 - \alpha} = 1 + \frac{1}{\beta}.$$

Since $\beta = [0;\, g(1), g(2), \ldots]$, its reciprocal is $1/\beta = [g(1);\, g(2), g(3), \ldots]$, and therefore

$$\frac{1}{1 - \alpha} = 1 + [g(1);\ g(2),\ g(3), \ldots] = [g(1) + 1;\ g(2),\ g(3), \ldots].$$

Hence $1 - \alpha = [0;\ g(1) + 1,\ g(2),\ g(3), \ldots]$.

In both cases $-x$ has integer part $-a_0 - 1$ and body generator g^* as stated. □

catena — `SimpleContinuedFraction.__neg__`

```
phi = SimpleContinuedFraction(lambda n: 1, integer_part=1)
# phi = [1; 1, 1, 1, ...] (golden ratio)
neg_phi = -phi
neg_phi.integer_part    # -2
neg_phi.segment(0)      # [-2;]
neg_phi.segment(1)      # [-2; 2]
neg_phi.segment(2)      # [-2; 2, 1]

sqrt5 = catena.numbers.constants.sqrt5
# sqrt(5) = [2; 4, 4, 4, ...]
neg_sqrt5 = -sqrt5
neg_sqrt5.integer_part    # -3
neg_sqrt5.segment(0)      # [-3;]
neg_sqrt5.segment(1)      # [-3; 1]
```

```
neg_sqrt5.segment(2)      # [-3; 1, 3]
neg_sqrt5.segment(5)      # [-3; 1, 3, 4, 4, 4]
```

Proposition 2.33 (Integer Translation)**.** *Let $x = (a_0, g)$ be an SCF and $n \in \mathbb{Z}$. Then*

$$x + n = (a_0 + n,\ g).$$

That is, adding an integer to an SCF shifts only the integer part, leaving the generator unchanged. In catena*, this is implemented via* `shift(n)`*, which creates a new instance sharing the same generator and convergent cache. Both* `scf + n` *and* `n + scf` *are supported; non-integer arguments return* `NotImplemented`*.*

catena — SimpleContinuedFraction.__add__

```
phi = SimpleContinuedFraction(lambda n: 1, integer_part=1)
# phi = [1; 1, 1, 1, ...] (golden ratio)
shifted = phi + 3         # [4; 1, 1, 1, ...]
shifted.integer_part      # 4
3 + phi                   # also [4; 1, 1, 1, ...] (__radd__ is
    symmetric)
# Only integer arguments are accepted:
# phi + 1.5               # returns NotImplemented
```

3 Finite Simple Continued Fractions

We now turn to the case of finite simple continued fractions, which correspond to rational numbers. As mentioned in the definition of the `SimpleContinuedFraction` class, the body generator g is allowed to have a finite domain, in which case the resulting SCF is finite.

Proposition 3.1 (Dual Representations of Rationals)**.** *Let $p/q \in \mathbb{Q}$. If $p/q \notin \mathbb{Z}$, it admits exactly two finite SCF representations:*

$$\frac{p}{q} = [a_0;\, a_1, \ldots, a_n] = [a_0;\, a_1, \ldots, a_{n-1},\, a_n - 1,\, 1],$$

where the first form has $a_n \geq 2$. If $p/q \in \mathbb{Z}$, the unique representation is $[p/q]$ (empty body).

The canonical form *is the first representation (with $a_n \geq 2$). Both representations are valid SCFs; the alternate form is obtained from the canonical one by replacing the last quotient a_n with $a_n - 1$ followed by 1.*

Proof sketch. When the Euclidean algorithm reaches its final non-zero remainder r_n, the last quotient is $a_n = r_{n-1}/r_n \geq 1$. If $a_n \geq 2$ we have the canonical form. If $a_n = 1$ we rewrite $[\ldots, 1]$ as $[\ldots]$ by absorbing the trailing 1 into $a_{n-1} + 1$, restoring $a_{n-1} + 1 \geq 2$; this is the unique canonical form. Conversely, starting from the canonical form one can always perform one extra Euclidean step: replace a_n by $a_n - 1$ and append 1, giving the alternate form. Since $a_n \geq 2$ the subtracted quotient $a_n - 1 \geq 1$ is a valid partial quotient. No third representation exists because any further step would require a quotient of 0. See [3] for a full proof. □

Remark 3.2. The constructor `FiniteSimpleContinuedFraction(pq, integer_part=a0)` validates that every element of `pq` is a strictly positive integer, but does *not* check whether the last element equals 1 (which would indicate a non-canonical form). The burden of canonical form is placed on the factory methods: `from_rational`, `from_float`, and `from_decimal` all delegate to `from_rational_to_scf` (which runs the Euclidean algorithm and naturally produces canonical output), so their result is always canonical. Direct construction via the constructor allows non-canonical inputs for flexibility.

3.1 The Euclidean Algorithm as SCF Expansion[10]

The SCF coefficients of p/q are precisely the successive quotients produced by the Euclidean algorithm. Concretely, `catena.mathlib.convert.from_rational_to_scf` implements:

$$\begin{aligned} r_0 &= p, \quad r_1 = q, \\ a_k &= \left\lfloor \frac{r_k}{r_{k+1}} \right\rfloor, \\ r_{k+2} &= r_k \bmod r_{k+1}, \qquad k \geq 0, \end{aligned}$$

until $r_{k+1} = 0$. The resulting sequence $(a_0, a_1, \ldots)$ is the SCF expansion of p/q.

Remark 3.3. This correspondence between the Euclidean algorithm and SCF expansion provides the bijection between finite (canonical) SCFs and rational numbers.

The library uses `divmod` (Python's built-in, backed by hardware division) for each step, relying on Python's arbitrary-precision integers so that very large rationals are handled correctly without overflow.

Note that r_k denotes the Euclidean remainders here, *not* the convergent numerators; the letter p (and h in the convergents section) is reserved for the latter throughout.

[10]The following description of the Euclidean algorithm is not a formal proof, but rather an explanation of how the algorithm corresponds to the SCF expansion process. For the representations of numbers as SCFs see Khinchin, §2, [3]. For a breakdown of the Euclidean algorithm into SCF steps see Parrish, §2.2, [4].

`from_rational_to_scf` returns the integer part and the body list as a pair $(a_0, [a_1, \ldots, a_n])$, matching the split representation used throughout the library:

catena — `from_rational_to_scf`

```
from catena.mathlib.convert import from_rational_to_scf

# Returns (integer_part, body_list)
from_rational_to_scf((355, 113))   # (3, [7, 16])
from_rational_to_scf((1, 1))       # (1, [])     -- pure integer, empty body
from_rational_to_scf((-3, 7))      # (-1, [1, 1, 3])
```

catena — Conversion from Rational-like Types to Finite SCF

```
from catena import FiniteSimpleContinuedFraction

# Convert a rational (Fraction | tuple[int, int] | int) number to its finite SCF representation.
rational_number = (17, 15) # Represents 17/15
finite_scf = FiniteSimpleContinuedFraction.from_rational(rational_number)
print(finite_scf) # FiniteSimpleContinuedFraction(partial_quotients=(7, 2), integer_part=1)

# Convert a float number to its finite SCF representation.
# Note: this conversion may not be exact due to the nature of floating-point representation.
float_number = 3.14
finite_scf_from_float = FiniteSimpleContinuedFraction.from_float(float_number)
print(finite_scf_from_float) # FiniteSimpleContinuedFraction(partial_quotients=(7, 7, 3216856876693, 14), integer_part=3)

finite_scf_from_float_limited = FiniteSimpleContinuedFraction.from_float(3.14, max_denominator=10000) # Limit the denominator to 10,000 to elude floating-point precision artifacts.
print(finite_scf_from_float_limited) # FiniteSimpleContinuedFraction(partial_quotients=(7, 7), integer_part=3)

# Convert a decimal string to its finite SCF representation.
finite_scf_from_decimal = FiniteSimpleContinuedFraction.from_decimal("3.14")
print(finite_scf_from_decimal) # FiniteSimpleContinuedFraction(partial_quotients=(7, 7), integer_part=3)

# from_decimal also accepts integer strings.
finite_scf_from_int_str = FiniteSimpleContinuedFraction.from_decimal("42")
print(finite_scf_from_int_str) # FiniteSimpleContinuedFraction(partial_quotients=(), integer_part=42)
```

Remark 3.4. The coefficient `3216856876693` appearing in the SCF of `from_float(3.14)` is not a property of the rational number 314/100, but of the IEEE 754 double-precision binary fraction nearest to 3.14. That value is $7070651414971679/2^{51}$, and its SCF expansion reflects the fine structure of this binary fraction. Use `from_decimal("3.14")` whenever the exact decimal interpretation is intended.

Remark 3.5. `FiniteSimpleContinuedFraction.from_decimal` takes a decimal string as input

rather than a `decimal.Decimal` object. This is because `Decimal` depends on a global precision context (`getcontext().prec`), and operations such as `Decimal(2).sqrt()` produce approximations whose accuracy depends on that context. Converting directly from `Decimal` would therefore require an additional precision parameter and would not produce consistent results across environments. Using a string avoids this ambiguity by making the input representation explicit.

Internally, `from_decimal` converts the string to an exact rational (p, q) via `from_decimal_to_rational` — splitting on the decimal point, computing $p = \text{integer_part} \cdot 10^d + \text{fractional_part}$ and $q = 10^d$ where d is the number of decimal digits, then reducing by $\gcd(p, q)$ — and then delegates to `from_rational`. This path is exact for every finite decimal string.

catena — `FiniteSimpleContinuedFraction.to_decimal`

```
scf = FiniteSimpleContinuedFraction.from_decimal("3.14")
scf.to_decimal()    # Decimal('3.14')

# Decimal(scf) does not work: Python has no __decimal__ protocol.
# Always call .to_decimal() explicitly.
```

3.2 Properties and Operations of Finite SCFs

The attributes of finite SCFs are mostly inherited from the general case, but several of them have special significance or different behaviour in the finite case. The table below lists the principal properties and methods provided by `FiniteSimpleContinuedFraction`:

Property / method	Description
`size`	Number of body elements k, equivalent to call `len(scf)`
`partial_quotients`	Full body $(a_1, \ldots, a_k)$ as a tuple
`terminal_convergent`	Exact rational value as (p, q)
`terminal_tail_convergent`	Last tail convergent (h_{k-1}, k_{k-1})
`to_decimal()`	Exact `Decimal` value
`float(scf)`	Float via terminal convergent
`bool(scf)`	`False` only for the zero SCF $[0;]$
`segment(n)`	Truncation to first n body elements
`scf_x + scf_y`	Rational sum as a new finite SCF
`scf.inverse()`	Rational inverse as a new finite SCF

Definition 3.6 (Terminal Convergent)**.** For a finite SCF $[a_0; a_1, \ldots, a_k]$ with $k =$ `size`, the *terminal convergent* is the convergent at index $k - 1$:

$$\frac{p}{q} = [a_0; a_1, \ldots, a_k].$$

This is the exact rational value of the SCF. By the coprimality corollary (Corollary 2.19), $\gcd(p, q) = 1$.

For the degenerate case $k = 0$ (empty body), the terminal convergent is $(a_0, 1)$.

catena — `FiniteSimpleContinuedFraction.terminal_convergent`

```
scf = FiniteSimpleContinuedFraction.from_rational((355, 113))
scf.terminal_convergent    # (355, 113)

scf_int = FiniteSimpleContinuedFraction([], integer_part=7)
scf_int.terminal_convergent  # (7, 1)
```

`terminal_convergent` is the central operation of `FiniteSimpleContinuedFraction`: it is called internally by `__float__`, `to_decimal`, `__add__`, and `inverse`.

Remark 3.7 (`float()` for finite vs. infinite SCFs)**.** For infinite SCFs, `float()` uses the 50th convergent as a heuristic approximation: the tail generator may not terminate, so there is no canonical last term to evaluate. For finite SCFs the terminal convergent is the exact rational value, so `float()` computes `terminal_convergent[0] / terminal_convergent[1]` directly — a single Python float division. No heuristic is needed or used.

Proposition 3.8 (Addition of Finite SCFs)**.** *Let $x = [a_0;\, a_1, \ldots, a_k]$ and $y = [b_0;\, b_1, \ldots, b_m]$ be finite SCFs with terminal convergents p_x/q_x and p_y/q_y respectively. Then*

$$x + y = \left[\frac{p_x q_y + p_y q_x}{q_x q_y}\right]_{\mathrm{SCF}},$$

where the right-hand side denotes the canonical SCF expansion of the rational sum.

catena — `FiniteSimpleContinuedFraction.__add__`

```
x = FiniteSimpleContinuedFraction.from_rational((3, 4))   # 3/4
y = FiniteSimpleContinuedFraction.from_rational((1, 6))   # 1/6
z = x + y   # 3/4 + 1/6 = 11/12
z.terminal_convergent   # (11, 12)

# Adding an integer shifts the integer part (inherited behaviour).
x + 2   # represents 3/4 + 2 = 11/4
```

Note that `scf + int` is handled by the parent class and shifts a_0 without touching the body; `scf + scf` is the proposition above and always returns a new `FiniteSimpleContinuedFraction` in canonical form.

Proposition 3.9 (Inverse of a Finite SCF)**.** *Let $x = [a_0;\, a_1, \ldots, a_k]$ with terminal convergent p/q, $p \neq 0$. Then*

$$\frac{1}{x} = \left[\frac{q}{p}\right]_{\mathrm{SCF}}.$$

Note the design difference from the parent class: `SimpleContinuedFraction.inverse()` manipulates the generator directly (prepending a_0 or shifting the index), which is appropriate when the generator is infinite. `FiniteSimpleContinuedFraction.inverse()` delegates to this same parent-class logic, so the result is always in canonical form without re-running the Euclidean algorithm (provided the original SCF is in canonical form). As an additional optimisation, if the terminal tail convergent is already cached on `self` at the time of the call, its value is transplanted into the inverse's convergent cache, so that a subsequent call to `inv.terminal_convergent` avoids recomputing the full recurrence. The inverse reference is memoised (write-once cache) so that calling `inverse()` twice returns the original object.

catena — `FiniteSimpleContinuedFraction.inverse`

```
x = FiniteSimpleContinuedFraction.from_rational((3, 4))   # 3/4
x.inverse().terminal_convergent   # (4, 3)

x.inverse().inverse() is x   # True  (memoised pair)

zero = FiniteSimpleContinuedFraction([], integer_part=0)
zero.inverse()   # raises ZeroDivisionError
```

3.3 Full Arithmetic on Finite SCFs

Since every finite SCF represents a rational number exactly (via `terminal_convergent`), the full field arithmetic of $\mathbb{Q}$ is available: addition, subtraction, multiplication, and division all reduce to the corresponding operation on the terminal fractions followed by a call to `from_rational`. The results are always canonical finite SCFs.

Remark 3.10 (Arithmetic strategy: terminal convergent first)**.** All four binary operations follow the same pattern:

1. Extract the terminal convergents (p_x, q_x) and (p_y, q_y).
2. Perform the rational arithmetic exactly in $\mathbb{Z}$.
3. Re-expand the result via `from_rational` (Euclidean algorithm), producing a canonical finite SCF.

This contrasts with the infinite case, where arithmetic must be implemented at the level of generators (e.g. `shift`, `prepend`, `advance`) because there is no terminal frame to extract.

The following table summarises which argument types are accepted by each binary operator. The result is always a `FiniteSimpleContinuedFraction` (or `NotImplemented` for unsupported types).

Operand type	`+`	`-`	`*`	`/`
`FiniteSimpleContinuedFraction`	✓	✓	✓	✓
`int`	†	✓	✓	✓
`float`	✓	✓	✓	✓
`Fraction`	✓	✓	✓	✓
`Decimal`	✓	✓	✓	✓

† `scf + int` is inherited from `SimpleContinuedFraction` and shifts only the integer part a_0, leaving the body generator unchanged (see Integer Translation in the previous section). All other combinations compute the rational result and re-expand via `from_rational`. For `float` and `Decimal` arguments the value is first converted to an exact `Fraction` before the rational arithmetic; any rounding already present in the float is preserved (not corrected).

Proposition 3.11 (Subtraction of Finite SCFs)**.** *Let x, y be finite SCFs. Then*

$$x - y = x + (-y),$$

where $-y$ is the additive inverse of y (see below). `__sub__` delegates entirely to `__add__` after negating the right-hand operand, so the same type dispatch applies.

Proposition 3.12 (Multiplication of Finite SCFs)**.** *Let x and y be finite SCFs with terminal convergents p_x/q_x and p_y/q_y. Then*

$$x \cdot y = \left[\frac{p_x\, p_y}{q_x\, q_y}\right]_{\text{SCF}}.$$

catena — `FiniteSimpleContinuedFraction.__mul__`

```
x = FiniteSimpleContinuedFraction.from_rational((3, 4))   # 3/4
y = FiniteSimpleContinuedFraction.from_rational((2, 3))   # 2/3
z = x * y   # 3/4 * 2/3 = 1/2
z.terminal_convergent   # (1, 2)

# Scalar multiplication
x * 3     # 3/4 * 3 = 9/4; terminal_convergent = (9, 4)
x * 0.5   # 3/4 * 1/2 = 3/8 (float converted to Fraction first)
```

Proposition 3.13 (Division of Finite SCFs)**.** *Let x and y be finite SCFs with terminal convergents p_x/q_x and p_y/q_y, with $p_y \neq 0$. Then*

$$\frac{x}{y} = \left[\frac{p_x\, q_y}{q_x\, p_y}\right]_{\text{SCF}}.$$

Note that $x/y = x \cdot (1/y)$, but catena *avoids constructing the intermediate inverse SCF; it operates directly on the four integers (p_x, q_x, p_y, q_y).*

catena — `FiniteSimpleContinuedFraction.__truediv__`

```
x = FiniteSimpleContinuedFraction.from_rational((3, 4))    # 3/4
y = FiniteSimpleContinuedFraction.from_rational((2, 3))    # 2/3
z = x / y    # (3/4) / (2/3) = 9/8
z.terminal_convergent    # (9, 8)

x / 2      # 3/4 / 2 = 3/8
```

Proposition 3.14 (Negation of a Finite SCF)**.** *Let $x = [a_0;\, a_1, \ldots, a_k]$ be a finite SCF with terminal convergent p/q. Then*

$$-x = \left[\frac{-p}{q}\right]_{\text{SCF}}.$$

For $k \leq 2$, the general negation formula from Proposition 2.32 requires accessing a_1 and a_2, which may not both exist. catena *therefore handles this case separately by delegating directly to* `from_rational(-Fraction(p, q))`*, and falls back to the generator-level formula for $k \geq 3$.*

catena — `FiniteSimpleContinuedFraction.__neg__`

```
x = FiniteSimpleContinuedFraction.from_rational((3, 4))    # [0; 1,
    3] = 3/4
(-x).terminal_convergent    # (-3, 4)

y = FiniteSimpleContinuedFraction.from_rational((355, 113))  # [3;
    7, 16]
(-y).terminal_convergent    # (-355, 113)
```

Remark 3.15 (Comparison with `SimpleContinuedFraction` arithmetic)**.** The table below summarises the key design differences between arithmetic on finite and infinite SCFs.

Operation	`SimpleContinuedFraction`	`FiniteSimpleContinuedFraction`
`+int`	shifts a_0, $O(1)$, no re-expansion	same (inherited)
`+scf`	not defined	terminal fracs → `from_rational`
`-`	not defined	delegates to `+(-other)`
`*`	not defined	terminal fracs → `from_rational`
`/`	not defined	terminal fracs → `from_rational`
`inverse`	generator-level prepend/shift	swap $(p, q) \to (q, p)$, re-expand
`neg`	generator-level case split	`from_rational` for $k \leq 2$, generator formula for $k \geq 3$
`float`	`convergent(50)` heuristic	exact `terminal[0]/terminal[1]`

The unifying principle is that finite SCFs are rational numbers, so all arithmetic is exact and the canonical form is always recoverable via the Euclidean algorithm. Infinite SCFs, by contrast, lack a terminal frame and must operate structurally at the generator level.

4 Caching Strategy

4.1 Motivation

Two separate caching concerns arise in `catena`.

Tail-convergent caching. The recurrence

$$h_n = g(n)\,h_{n-1} + h_{n-2}, \quad k_n = g(n)\,k_{n-1} + k_{n-2}$$

has optimal substructure: h_n depends on h_{n-1} and h_{n-2}. Without memoisation a naive recursive implementation recomputes all earlier tail convergents each time a later one is requested. With memoisation each value is computed once, reducing the total cost from $O(2^n)$ to $O(n)$. All `SimpleContinuedFraction` instances carry a dedicated `_tail_cache` for this purpose.

Generator caching. When the partial-quotient function $g : \mathbb{N} \to \mathbb{Z}^+$ is expensive to evaluate (e.g. a number-theoretic function or a transcendental constant computed digit by digit), re-evaluating $g(n)$ on every recurrence step would dominate the runtime. `CachedGenerator` wraps an arbitrary callable and memoises each distinct index in a `BaseCache`.

4.2 The `BaseCache` Primitive

`catena`'s caching layer is built around a single primitive defined in `catena.cache`: `BaseCache`. It is a self-computing, append-only dictionary (`UserDict` subclass) that unifies the three separate classes that existed in earlier versions (`Cache`, `OrdinalCache`, and `CacheHandler`).

Immutability after write.
Once a key is stored, any subsequent attempt to overwrite it raises `KeyError`. This is correct for convergents and generator values alike, because both are deterministic functions of their index for a fixed SCF.

Self-computing on miss.
`BaseCache` is constructed with a bound callable `func`. Accessing a missing key via `cache[key]` calls `func(key)`, stores the result, and returns it — no external decorator or wrapper is needed. Explicit writes (`cache[key] = value`) are also accepted and bypass the function call.

Key-range tracking.
For integer keys, `smallest_key` and `largest_key` are maintained without iterating the dictionary. The `largest_key` sentinel drives the tail-convergent iterative fill: the recurrence advances from `largest_key + 1` to n, computing only the truly missing entries in $O(\text{gap})$ time and $O(1)$ call-stack depth regardless of n.

Usage statistics.
`call_count` counts cache misses (calls to `func`), `read_count` counts cache hits (stored-result retrievals), and `write_count` counts all writes (including those from seeding and intermediate fills).

Capacity management.
An optional `maxsize` parameter caps the number of stored entries. When a new write would exceed the limit, `prune(n, order)` is called automatically using a configurable `prune_key` callable. The default policy retains the `maxsize - 1` largest-keyed entries, keeping the recurrence frontier intact. `reset()` clears the cache and all counters; `clear()` clears entries only, preserving `call_count` and `read_count`.

Construction-time seeding.
The `seed` constructor parameter pre-populates the cache from a plain dict without incrementing any statistics counters. This is the mechanism used by `CachedGenerator.advance` and `CachedGenerator.insert` to carry relevant entries across cache-shifting operations.

Blocked mutation methods.
The mutating `UserDict` methods (`pop`, `popitem`, `__delitem__`, `update`, `setdefault`, `fromkeys`, `__or__`) are all disabled and raise `NotImplementedError`. Use `prune` or `reset` to reduce cache size.

4.3 Generator-Level Caching: `CachedGenerator`

`CachedGenerator` is a subclass of `Generator` that internally holds a `BaseCache` bound to the wrapped callable. Every call `cgen(n)` is routed through the cache: on a miss the underlying function is invoked and the result stored; on a hit the stored result is returned directly.

When to use each generator type.

1. `Generator`: the default. Wrap any callable $g : \mathbb{N} \to \mathbb{Z}^+$ that is cheap to evaluate (e.g. a closed-form formula).

2. `CachedGenerator`: use when $g(n)$ is expensive *and* catena's cache introspection is needed (`call_count`, `read_count`, `stats`, `BaseCache` bookkeeping). Also the right choice when one expensive generator must be shared across several SCF instances so that all instances draw from the same memoised pool.

3. Sharing via `Generator` wrapping an `@lru_cache`-decorated function achieves the same functional sharing without catena's introspection layer. Use this pattern for recursive functions (Fibonacci, etc.): apply `@functools.lru_cache` to the function itself *before* passing it to `Generator` or `CachedGenerator`, because `CachedGenerator` only intercepts top-level calls and does not propagate into self-recursive calls inside the function body.

4. `FiniteGenerator`: use for fixed finite sequences; no caching is needed because lookup is $O(1)$ by array index.

catena — Caching Patterns

```
from catena.generators import Generator, CachedGenerator
from catena import SimpleContinuedFraction
import functools, math

# --- Pattern 1: expensive non-recursive generator ---
# g(n) = number of divisors of (n+2) not exceeding sqrt(n+2)
def small_divisors(n):
    m = n + 2
    return sum(1 for d in range(1, math.isqrt(m) + 1) if m % d == 0)

cgen = CachedGenerator(small_divisors)
scf = SimpleContinuedFraction(cgen, integer_part=0)
scf.convergent(20)       # fills cache entries 0..20 as a side-
   effect
cgen.call_count          # 21  (one miss per index)
cgen.read_count          # 0   (no hits yet from this SCF)

# --- Pattern 2: sharing one cache across multiple SCFs ---
scf2 = SimpleContinuedFraction(cgen, integer_part=1)
scf2.convergent(10)      # cgen(0)..cgen(10) are already cached
cgen.read_count          # 11  (all hits: entries 0..10 were
```

```
    populated by scf)

# --- Pattern 3: recursive generator - pre-memo with lru_cache ---
@functools.lru_cache(maxsize=None)
def fib(n):
    return fib(n - 1) + fib(n - 2) if n > 1 else n + 1

# Wrap in CachedGenerator to add BaseCache introspection on top
cgen_fib = CachedGenerator(fib)     # fib's own lru_cache handles
    recursion
cgen_fib(10)                        # fib(10) was already cached by
    lru_cache
cgen_fib.call_count                 # 1
cgen_fib.cache.reset()              # clears BaseCache; lru_cache
    unchanged
```

4.4 Shared Caches and the Tail

All `tail()`, `shift()`, and `segment(size)` operations create a new SCF instance via `_from_shared`, which copies the *reference* to `_tail_cache` rather than duplicating the cache object. Consequently both the original and the derived SCF read from and write to the same `BaseCache`; a tail-convergent computed by one is immediately available to the other at zero cost.

This is the mechanism that makes the following hold:

catena — Shared tail-convergent cache

```
from catena import SimpleContinuedFraction

phi = SimpleContinuedFraction(lambda n: 1, integer_part=1)
phi.tail_convergent(9)              # populates cache entries 0..9

tail = phi.tail()                   # new instance, same _tail_cache
tail.cache_handler is phi.cache_handler   # True
tail.tail_convergent(9)             # O(1) cache hit -- no
    recomputation
phi.cache_handler.read_count        # incremented by the hit above
```

5 Periodic Simple Continued Fractions

By Lagrange's theorem, a real number has an eventually periodic continued fraction expansion if and only if it is a *quadratic irrational* or *quadratic surd* — a root of a degree-two polynomial with integer coefficients that is not itself rational. The favorite examples are $\sqrt{2} = [1; \overline{2}]$, $\varphi = [1; \overline{1}]$, and more generally every expression of the form $(P + \sqrt{D})/Q$ with D not a perfect square. This section introduces `PeriodicSimpleContinuedFraction`, the `catena` class that represents such numbers exactly and exposes their structure through the same functional interface used by `SimpleContinuedFraction`.

5.1 Purely periodic vs. eventually periodic

Definition 5.1 (Eventually periodic SCF)**.** An infinite SCF (a_0, g) with $g : \mathbb{N} \to \mathbb{Z}^+$ is *eventually periodic* if there exist integers $m \geq 0$ and $r \geq 1$ such that

$$g(m + k) = g(m + k + r) \qquad \text{for all } k \geq 0.$$

The smallest such r is the *minimal period length* and the corresponding block $(g(m), \ldots, g(m + r - 1))$ is the *period*. The block $(g(0), \ldots, g(m - 1))$ (empty when $m = 0$) is the *pre-period*. When $m = 0$ the SCF is *purely periodic* (its body repeats from the very first partial quotient).

In `catena` there is no distinction between the two at the type level. Both are represented by a single class, `PeriodicSimpleContinuedFraction`, instantiated with the same constructor. The difference is entirely encoded by the pre-period length m: a purely periodic PSCF is simply one where $m = 0$, so that g_{pre} has an empty domain and contributes no partial quotients. In Definition 5.4 this means the case-split collapses to $g(n) = g_{\text{per}}(n \bmod r)$ for all $n \geq 0$, and the `pre_period` argument to the constructor is omitted (or passed as an empty sequence).

Remark 5.2. Some sources define "purely periodic" to include a_0 in the repeating block, writing $[\overline{a_0;\, a_1, \ldots, a_{r-1}}]$. In `catena` the integer part a_0 is always stored separately and is never part of a periodic block. "Purely periodic" here refers to the body g alone: $m = 0$ means the body repeats from index 0, regardless of the value of a_0.

5.2 The integer-part separation and its effect on the period

Because a_0 is stored outside the generator, the period reported by `PeriodicSimpleContinuedFraction` is always the period of the *body generator* g, not of the full sequence $(a_0, a_1, a_2, \ldots)$. This can shift the apparent repeating block relative to a naive reading of the expansion.

Example 5.3. Consider $x = [3; \overline{1, 3}]$. Written out, the partial quotients are $3, 1, 3, 1, 3, \ldots$ The digits 3 and 1 alternate, so one might expect the period to be $(3, 1)$. However, $a_0 = 3$ is the integer part and is stored separately; the body sequence is $1, 3, 1, 3, \ldots$, which has period $(1, 3)$. The number $[\overline{3, 1}]$ (in the sense of a purely periodic expansion starting from a_0) is the same value — the two representations differ only in where the period is "cut" — but `catena` always extracts $a_0 = \lfloor x \rfloor$ first, so the stored period is $(1, 3)$ in both cases.

This is not a limitation but a deliberate consequence of the uniform representation: the integer part is always an integer, the body generator always maps into $\mathbb{Z}^+$, and the two are never mixed.

5.3 Definition and data model

Definition 5.4 (Periodic simple continued fraction)**.** A *periodic simple continued fraction* (PSCF) is an eventually periodic SCF (a_0, g) whose body generator $g : \mathbb{N} \to \mathbb{Z}^+$ is determined by two

finite generators

$$g_{\text{pre}} : m \longrightarrow \mathbb{Z}^+,$$
$$g_{\text{per}} : r \longrightarrow \mathbb{Z}^+, \quad r \geq 1,$$

via the rule

$$g(n) \;=\; \begin{cases} g_{\text{pre}}(n) & \text{if } n < m, \\ g_{\text{per}}((n-m) \bmod r) & \text{if } n \geq m. \end{cases}$$

Here $m \in \mathbb{N}$ is the *pre-period length* and $r \in \mathbb{Z}^+$ is the *period length*. The sequences

$$(g_{\text{pre}}(0), \ldots, g_{\text{pre}}(m-1)) \qquad \text{and} \qquad (g_{\text{per}}(0), \ldots, g_{\text{per}}(r-1))$$

are referred to as the *pre-period* and the *period* of the PSCF respectively.

In `catena`, g_{pre} and g_{per} are each stored as a `FiniteGenerator` — a fixed, immutable array of positive integers backed by a compact `array.array`. These two `FiniteGenerator` instances are then combined into a single `PeriodicGenerator`, which implements the case-split of Definition 5.4 as its underlying callable. `PeriodicSimpleContinuedFraction` wraps this generator together with the integer part a_0, and exposes the two sub-generators through the `non_repeating_part` and `period` properties.

catena — PeriodicSimpleContinuedFraction

```
from catena import PeriodicSimpleContinuedFraction as PSCF

# phi = [1; 1,1,1,...] -- golden ratio, purely periodic body (m = 0)
phi = PSCF(period=[1], integer_part=1)
print(phi.integer_part)        # 1
print(phi.period)              # (1,)
print(phi.non_repeating_part)  # ()

# sqrt(2) = [1; 2,2,2,...] -- purely periodic body (m = 0)
sqrt2 = PSCF(period=[2], integer_part=1)

# [3; 1,3,1,3,...] -- the body period is (1,3), not (3,1)
x = PSCF(period=[1, 3], integer_part=3)
print(x.period)               # (1, 3)
print(x.non_repeating_part)  # ()

# (7 + sqrt(10))/4 = [2; 1,1, 5,1,1,1,24,1,1,1,5,...]
# non-repeating pre-period (1,1) followed by period
   (5,1,1,1,24,1,1,1)
y = PSCF(period=[5, 1, 1, 1, 24, 1, 1, 1], pre_period=[1, 1],
   integer_part=2)
print(y.non_repeating_part)  # (1, 1)
print(y.period)              # (5, 1, 1, 1, 24, 1, 1, 1)
```

5.4 Lagrange's theorem and quadratic surds

Lagrange's theorem states that a real number has an eventually periodic SCF expansion if and only if it is a quadratic irrational. The proof is not difficult, but it falls outside the scope of this document. Nonetheless, the theorem is fundamental to the design of `PeriodicSimpleContinuedFraction` and to the correctness of its methods, so we will not only state it here but also use it as a guiding principle. The two constructive directions of this correspondence — expanding a given $(P + \sqrt{D})/Q$ into its PSCF, and recovering the surd parameters from a PSCF — are developed in §5.5.

Definition 5.5 (Quadratic surd)**.** A *quadratic surd* is a real number of the form

$$x = \frac{P + \sqrt{D}}{Q},$$

where $P, Q, D \in \mathbb{Z}$, $Q \neq 0$, $D \geq 0$, and D is not a perfect square.

Remark 5.6. The term "quadratic irrational" is often used interchangeably with "quadratic surd", but the latter emphasizes the radical representation. In the context of continued fractions, both terms refer to the same class of numbers that have eventually periodic expansions.

Moreover, it is not infrequent to encounter the equivalent form $(a + b\sqrt{d})$ with $a, b \in \mathbb{Q}$ and $d \in \mathbb{Z}^+$ not a perfect square. The choice of representation is a matter of convention, and the two forms are related by simple algebraic manipulations. In catena, we adopt the $(P + \sqrt{D})/Q$ form for consistency and to facilitate the implementation of methods that rely on the algebraic properties of quadratic surds.

The triple (P, Q, D) is not a unique representation of a given quadratic surd: for example, $(2, 4, 8)$ and $(1, 2, 2)$ both represent $\frac{1+\sqrt{2}}{2}$ via $(2 + \sqrt{8})/4 = (2 + 2\sqrt{2})/4 = (1 + \sqrt{2})/2$. The two normalization operations below address distinct aspects of this non-uniqueness.

5.4.1 Simplified form

Definition 5.7 (Simplified form)**.** A quadratic surd $(P + \sqrt{D})/Q$ is in *simplified form* if there is no integer $g > 1$ such that $g \mid P$, $g \mid Q$, and $g^2 \mid D$ simultaneously. Equivalently, writing s for the largest integer with $s^2 \mid D$, the condition is

$$\gcd(P,\, Q,\, s) = 1.$$

The simplification does *not* constrain the sign of Q.

Remark 5.8 (Motivation)**.** Simplified form is the canonical storage format for quadratic surds in catena. Without it, the same surd can be represented by infinitely many triples (kP, kQ, k^2D) for any non-zero integer k, making equality checks, hashing, and memoisation unreliable. By reducing to simplified form, every quadratic surd has a unique representation up to the sign of Q — a sign that carries separate semantic meaning (see principal vs. conjugate below) and must therefore be preserved.

Note that this is strictly a *representational* normalisation: it does not affect the sign of either P or Q, so $\operatorname{sign}(Q)$ is left intact.

catena — simplify_quadratic_surd

```
from catena.mathlib.quadratic import simplify_quadratic_surd

# (2 + sqrt(8)) / 4  =  (1 + sqrt(2)) / 2
simplify_quadratic_surd(2, 4, 8)    # (1, 2, 2)

# Signs are preserved: (-1 + sqrt(3)) / (-2) stays negative-Q
simplify_quadratic_surd(-1, -2, 3)  # (-1, -2, 3)  (already
    simplified)

# g = gcd(P, Q, square_part(sqrt(D))); divides P -> P/g, Q -> Q/g, D
    -> D/g^2
simplify_quadratic_surd(6, 4, 12)   # g = gcd(6, 4, 2) = 2 -> (3, 2,
    3)
```

5.4.2 Normalized form

Definition 5.9 (Normalized form)**.** A quadratic surd $(P + \sqrt{D})/Q$ is in *normalized form* if

$$Q \mid (D - P^2).$$

Remark 5.10 (Motivation)**.** Normalized form is an *algorithmic* requirement, not a canonicalization. The expansion algorithm `from_quadratic_surd_to_scf` computes successive remainders via the recurrence

$$m_{n+1} = a_n\, d_n - m_n, \qquad d_{n+1} = \frac{D - m_{n+1}^2}{d_n},$$

where $(m_0, d_0) = (P, Q)$ and $a_n = \lfloor (m_n + \lfloor \sqrt{D} \rfloor)/d_n \rfloor$. For d_{n+1} to be an integer, we need $d_n \mid (D - m_{n+1}^2)$ at every step. One can verify that this divisibility propagates automatically once it holds at $n = 0$, i.e., once $Q \mid (D - P^2)$.

When the user supplies an arbitrary triple (P, Q, D), this condition need not hold. `normalize_quadratic_sur` finds the minimal positive integer

$$k = \frac{|Q|}{\gcd(|Q|,\, |D - P^2|)}$$

and rescales $(P, Q, D) \mapsto (kP, kQ, k^2 D)$. The value is unchanged since $(kP + \sqrt{k^2 D})/(kQ) = (P + \sqrt{D})/Q$, but now $kQ \mid (k^2 D - k^2 P^2) = k^2(D - P^2)$, which follows because $k|Q|/k = |Q|$ divides $k(D - P^2) = \operatorname{lcm}(|Q|, |D - P^2|)$ by construction. The absolute value on Q is necessary to keep k positive (and hence preserve $\operatorname{sign}(Q)$ under the rescaling); the absolute value on $D - P^2$ is necessary so that the integer division $k = \operatorname{lcm}/|D - P^2|$ yields a positive result.

Simplified form and normalized form are independent: a surd can be simplified but not normalized (e.g. $(1, 3, 2)$, where $3 \nmid 2 - 1 = 1$) or normalized but not simplified (e.g. $(2, 4, 8)$, where $4 \mid 8 - 4 = 4$, but $g = 2 > 1$). In `catena`, normalization is applied internally by `from_quadratic_surd_to_scf` and is never exposed as a public state.

```
catena — normalize_quadratic_surd
from catena.mathlib.quadratic import normalize_quadratic_surd

# (1 + sqrt(2)) / 3: D - P^2 = 1, lcm(3,1)/1 = 3, so k = 3.
normalize_quadratic_surd(1, 3, 2)   # (3, 9, 18): 9 | (18 - 9) = 9
    (ok)

# Already normalized: (1 + sqrt(3)) / 2, D - P^2 = 2, 2 | 2 (ok), k
    = 1.
normalize_quadratic_surd(1, 2, 3)   # (1, 2, 3)
```

5.4.3 Quadratic polynomials and the minimal polynomial

Lemma 5.11 (Mapping between quadratic surds and quadratic polynomials)**.** *Every quadratic surd $(P + \sqrt{D})/Q$ (in simplified form) is the root of a unique simplified quadratic polynomial $Ax^2 + Bx + C = 0$ with integer coefficients and $A > 0$. Conversely, every such polynomial with positive non-square discriminant has exactly two roots, both quadratic surds, forming a conjugate pair.*

Proof. The roots of $Ax^2 + Bx + C = 0$ are $(-B \pm \sqrt{B^2 - 4AC})/(2A)$. Setting $P = -B$, $Q = 2A$, $D = B^2 - 4AC$ gives the surd form for the $+$ branch.

Conversely, given $(P + \sqrt{D})/Q$, we apply the normalized transformation $(P, Q, D) \mapsto (P^*, Q^*, D^*)$, where we have by construction $Q^* \mid (D^* - P^{*2})$, and the identity

$$\frac{P + \sqrt{D}}{Q} = \frac{P^* + \sqrt{D^*}}{Q^*}.$$

Then, multiplying

$$Q^*x - P^* = \sqrt{D^*}$$

through by itself gives

$$Q^{*2}x^2 - 2P^*Q^*x + (P^{*2} - D^*) = 0.$$

Since the surd is normalized, then dividing by Q^* yields

$$Q^*x^2 - 2P^*x + \frac{P^{*2} - D^*}{Q^*} = 0,$$

which has integer coefficients. Finally, we multiply through by $\text{sign}(Q^*)$ to ensure a positive leading coefficient. □

Definition 5.12 (Minimal polynomial and quadratic coefficients)**.** The *minimal polynomial* of a quadratic surd x is the unique polynomial $Ax^2 + Bx + C = 0$ with integer coefficients, $A > 0$, and $\gcd(A, B, C) = 1$. The triple (A, B, C) is called the *quadratic coefficients* of x.

Remark 5.13 (Why $A > 0$)**.** The sign convention $A > 0$ is a normalization choice. Any minimal polynomial can be multiplied by -1 without changing its roots, so without a convention the triple (A, B, C) is determined only up to sign. Fixing $A > 0$ makes the triple canonical, which is essential for hashing and equality: `PeriodicSimpleContinuedFraction.__eq__` and `__hash__` delegate to `quadratic_surd()`, which is itself derived from the canonical triple.

Furthermore, $A > 0$ ensures a consistent correspondence with the surd: the + branch of the quadratic formula, $(-B + \sqrt{B^2 - 4AC})/(2A)$, always gives the *larger* of the two roots when $A > 0$, which `catena` designates the *principal* root (see below).

Remark 5.14 (Quadratic coefficients of a PSCF with non-zero integer part)**.** `PeriodicSimpleContinuedFracti` first computes the minimal polynomial (A, B, C) of the tail SCF (the value of the *body generator* with $a_0 = 0$) via `PeriodicGenerator.quadratic_coefficients()`. If the stored integer part $a_0 = 0$, this is returned directly.

When $a_0 \neq 0$, the full value is $x = y + a_0$ where y satisfies $Ay^2 + By + C = 0$. Substituting $y = x - a_0$:

$$A(x - a_0)^2 + B(x - a_0) + C = 0 \quad \Longrightarrow \quad Ax^2 + (B - 2Aa_0)x + (Aa_0^2 - Ba_0 + C) = 0.$$

The adjusted coefficients are therefore

$$A' = A, \qquad B' = B - 2Aa_0, \qquad C' = C - Ba_0 + Aa_0^2.$$

Since $A > 0$ is preserved under this substitution and $\gcd(A, B, C) = 1$ also propagates, no further normalization is needed.

catena — PeriodicSimpleContinuedFraction.quadratic_coefficients

```
from catena import PeriodicSimpleContinuedFraction as PSCF

sqrt2 = PSCF(period=[2], integer_part=1)        # [1; 2,2,2,...] = sqrt(2)
sqrt2.quadratic_coefficients()    # (1, 0, -2)  ->  x^2 - 2 = 0

phi = PSCF(period=[1], integer_part=1)           # [1; 1,1,1,...] = golden ratio
phi.quadratic_coefficients()      # (1, -1, -1)  ->  x^2 - x - 1 = 0

# PSCF with non-zero integer part: y satisfies A'y^2 + B'y + C' = 0
# where the shift x = y + a0 is accounted for.
x = PSCF(period=[1, 3], integer_part=3)          # [3; 1,3,1,3,...]
```

```
x.quadratic_coefficients()        # the generator's (A,B,C) shifted
    by a0 = 3
```

5.4.4 Principal and conjugate surds

Definition 5.15 (Principal and conjugate roots)**.** Let x be a quadratic surd with minimal polynomial $Ax^2 + Bx + C = 0$ ($A > 0$). Setting $P = -B$, $Q = 2A$, $D = B^2 - 4AC$, the two roots are

$$x_+ = \frac{P + \sqrt{D}}{Q}, \qquad x_- = \frac{P - \sqrt{D}}{Q} = \frac{-P + \sqrt{D}}{-Q}.$$

Since $A > 0$ we have $Q > 0$, so x_+ is the *principal root* and x_- is the *conjugate root*. In catena's (P, Q, D) representation:

- $Q > 0 \iff$ the surd is the principal root (`is_principal_surd()`).
- $Q < 0 \iff$ the surd is the conjugate root (`is_conjugate_root()`).

Remark 5.16 (Sign of Q as a root selector)**.** The sign of Q is the only way catena distinguishes the two roots of the same minimal polynomial, without carrying an explicit $\pm$ flag. This design choice propagates through several operations:

- `quadratic_surd()` checks which root the stored PSCF expansion corresponds to (by re-expanding and comparing) and returns (P, Q, D) with the appropriate sign.
- `__neg__` maps $(P, Q, D) \mapsto (P, -Q, D)$, because $-(P + \sqrt{D})/Q = (P + \sqrt{D})/(-Q)$; this flips the root selector.
- `inverse()` rationalizes differently depending on the sign of Q (see below), because the denominators $P^2 - D$ may differ in sign between the two branches.

catena — is_principal_surd, is_conjugate_root

```
from catena import PeriodicSimpleContinuedFraction as PSCF

phi = PSCF(period=[1], integer_part=1)   # golden ratio phi ~ 1.618
phi.is_principal_surd()    # True   (Q > 0)
phi.is_conjugate_root()    # False

conj = phi.conjugate()     # the other root ~ -0.618
conj.is_principal_surd()   # False
conj.is_conjugate_root()   # True   (Q < 0)

# Negating flips the root selector
neg_phi = -phi             # -phi ~ -1.618
neg_phi.is_principal_surd()   # False  (sign of Q is flipped by
    negation)
```

Definition 5.17 (Algebraic conjugate)**.** Let $x = (P + \sqrt{D})/Q$ be a quadratic surd. Its *algebraic conjugate* is

$$\bar{x} = \frac{P - \sqrt{D}}{Q} = \frac{-P + \sqrt{D}}{-Q}.$$

In (P, Q, D) coordinates, passing to the conjugate sends $(P, Q, D) \mapsto (-P, -Q, D)$: the radicand D is unchanged, while both P and Q are negated.

Remark 5.18 (Why the conjugate matters)**.** The two roots x and $\bar{x}$ satisfy the same minimal polynomial, and by Galois theory every polynomial identity with rational coefficients that holds for one root also holds for the other. In the SCF setting, the conjugate appears naturally when inverting a PSCF: if $x = (P + \sqrt{D})/Q$ then

$$\frac{1}{x} = \frac{Q}{P + \sqrt{D}} = \frac{Q(P - \sqrt{D})}{P^2 - D} = \frac{Q^2}{P^2 - D}\bar{x},$$

i.e. the inverse is expressed using the conjugate's numerator. `catena` stores the conjugate in a write-once `_conjugate` attribute (mirroring the `_inverse` pattern) so that `conjugate()` is an involution: `scf.conjugate().conjugate() is scf`.

catena — PeriodicSimpleContinuedFraction.conjugate

```
from catena import PeriodicSimpleContinuedFraction as PSCF

phi = PSCF(period=[1], integer_part=1)       # phi = (1 + sqrt(5))/2
      ~  1.618
conj = phi.conjugate()                        # (1 - sqrt(5))/2  ~
   -0.618

phi.quadratic_surd()          # (1, 2, 5)    -> P=1,  Q=2  (principal)
conj.quadratic_surd()         # (-1, -2, 5)  -> P=-1, Q=-2 (conjugate)

phi.conjugate().conjugate() is phi    # True  (write-once involution)

# Both roots share the same minimal polynomial:
phi.quadratic_coefficients()     # (1, -1, -1)
conj.quadratic_coefficients()    # (1, -1, -1)
```

5.4.5 Multiplicative inverse

Lemma 5.19 (Reflected polynomial)**.** *Let x be a quadratic surd with minimal polynomial $Ax^2 + Bx + C = 0$ ($A > 0$, $\gcd(A, B, C) = 1$, $C \neq 0$). Then $y = 1/x$ is also a quadratic surd, satisfying*

$$Cy^2 + By + A = 0.$$

Normalizing to a positive leading coefficient gives the minimal polynomial of $1/x$:

$$(A',\, B',\, C') \;=\; \operatorname{sign}(C) \cdot (C,\, B,\, A).$$

We call this the reflected polynomial *of $Ax^2 + Bx + C = 0$. Its discriminant equals that of the original: $B^2 - 4A'C' = B^2 - 4AC = D$, so both polynomial families share the same radicand D.*

Proof. Multiply $Ax^2 + Bx + C = 0$ through by x^{-2}:

$$A + Bx^{-1} + Cx^{-2} = 0 \implies Cy^2 + By + A = 0, \quad y = 1/x.$$

Since $\gcd(A, B, C) = 1$ we also have $\gcd(C, B, A) = 1$, so multiplying by $\operatorname{sign}(C)$ produces produces the unique primitive polynomial with positive leading coefficient. The discriminant claim follows immediately from $B^2 - 4(\operatorname{sign}(C) \cdot C)(\operatorname{sign}(C) \cdot A) = B^2 - 4AC$. □

Remark 5.20 (Inversion formula and the role of $\operatorname{sign}(Q)$)**.** Although the algebraic identity

$$\frac{1}{x} = \frac{-QP + Q\sqrt{D}}{D - P^2}$$

holds formally, catena does not store arbitrary signed multiples of $\sqrt{D}$. By convention, quadratic surds are always represented in the form

$$\frac{P+\sqrt{D}}{Q},$$

with the choice of root encoded solely by $\operatorname{sign}(Q)$. Consequently, the sign of Q must be extracted explicitly from the radical term when rationalizing, leading to the two implementation branches below.

Starting from $x = (P+\sqrt{D})/Q$, rationalising gives

$$\frac{1}{x} = \frac{Q\,(P-\sqrt{D})}{P^2-D}.$$

To rewrite this in $(P'+\sqrt{D})/Q'$ form we need to extract $\sqrt{D}$ from the numerator. Because $\sqrt{Q^2D} = |Q|\sqrt{D}$, we have $Q\sqrt{D} = \operatorname{sign}(Q)\cdot\sqrt{Q^2D}$, and the two branches of the computation are therefore:

- $Q > 0$ **(principal surd):** $Q\sqrt{D} = \sqrt{Q^2D}$, so

  $$\frac{1}{x} = \frac{-QP+\sqrt{Q^2D}}{D-P^2},$$

  giving $(P',\,Q',\,D') = (-QP,\;D-P^2,\;Q^2D)$ before simplification.

- $Q < 0$ **(conjugate surd):** $Q\sqrt{D} = -\sqrt{Q^2D}$, so

  $$\frac{1}{x} = \frac{QP+\sqrt{Q^2D}}{P^2-D},$$

  giving $(P',\,Q',\,D') = (QP,\;P^2-D,\;Q^2D)$ before simplification.

In both cases the unsimplified triple is passed to `from_quadratic_surd`, which simplifies it and runs the expansion algorithm. The sign of the resulting Q' is then determined by checking which root of the reflected polynomial the PSCF actually represents (via `quadratic_surd()`), not by the intermediate arithmetic. Note that whether the inverse turns out to be the principal or conjugate root of the reflected polynomial depends on the sign of $D-P^2$: for the principal-surd branch, Q' inherits the sign of $D-P^2$ (before simplification), which is not fixed.

Remark 5.21 (Four quadratic surds related by conjugation and inversion)**.** Every quadratic surd x (with $C \neq 0$, so that $1/x$ is also irrational) belongs to a family of four objects closed under conjugation and inversion:

$$\begin{array}{ll} x & \bar{x} = \operatorname{conj}(x) \\ 1/x = \operatorname{inv}(x) & 1/\bar{x} = \operatorname{conj}(\operatorname{inv}(x)) \end{array}$$

The top row satisfies the minimal polynomial $Ax^2+Bx+C=0$; the bottom row satisfies the reflected polynomial (A', B', C') from Lemma 5.19. All four surds share the same radicand D.

The two operations commute:

$$\operatorname{inv}(\operatorname{conj}(x)) = \operatorname{conj}(\operatorname{inv}(x)),$$

so the family is the orbit of x under the group $\{\mathrm{id},\ \mathrm{conj},\ \mathrm{inv},\ \mathrm{inv}\circ\mathrm{conj}\} \cong (\mathbb{Z}/2\mathbb{Z})^2$. The sign of Q — catena's root selector — distinguishes the two elements in each row: $Q > 0$ for a principal root, $Q < 0$ for a conjugate root.

```
catena — PeriodicSimpleContinuedFraction.inverse
from catena import PeriodicSimpleContinuedFraction as PSCF

phi = PSCF(period=[1], integer_part=1)       # phi = (1 + sqrt(5))/2 ~ 1.618
# phi satisfies x^2 - x - 1 = 0
phi.quadratic_coefficients()                  # (1, -1, -1)

# 1/phi satisfies the reflected polynomial: swap A and C, normalize sign
# C = -1  ->  sign(C) * (C, B, A) = (-1)*(-1, -1, 1) = (1, 1, -1)  ->  y^2 + y - 1 = 0
inv_phi = phi.inverse()                       # (sqrt(5) - 1)/2  ~ 0.618
inv_phi.quadratic_coefficients()              # (1, 1, -1)

# The four related objects all share radicand D = 5:
phi.quadratic_surd()                          # (1, 2, 5)  principal of x^2 - x - 1 = 0
phi.conjugate().quadratic_surd()              # (-1, -2, 5)  conjugate of x^2 - x - 1 = 0
inv_phi.quadratic_surd()                      # (-1, 2, 5)  principal of y^2 + y - 1 = 0
inv_phi.conjugate().quadratic_surd()          # (1, -2, 5)  conjugate of y^2 + y - 1 = 0

# Commutativity of conjugate and inverse:
phi.conjugate().inverse() == inv_phi.conjugate()   # True

# Write-once involution:
phi.inverse().inverse() is phi                     # True
```

Remark 5.22 (Inversion does not preserve root orientation)**.** Unlike conjugation and negation, inversion does not act predictably on catena's root selector. If

$$x = \frac{P + \sqrt{D}}{Q}$$

is represented with $Q > 0$ (a principal surd), then before simplification the inverse has denominator

$$Q' = D - P^2.$$

Consequently, the sign of Q' — and therefore whether $1/x$ is principal or conjugate — depends on the arithmetic relation between P^2 and D. In particular, inversion may preserve or flip the root orientation.

5.5 Conversion algorithms: quadratic surd ↔ PSCF

Lagrange's theorem guarantees a bijective correspondence between quadratic surds and eventually-periodic SCFs, but the theorem itself is non-constructive. This subsection makes the correspondence explicit in both directions. §5.5.1 describes the algorithm that expands a given quadratic surd $(P + \sqrt{D})/Q$ into a PSCF; §5.5.2 describes the inverse recovery. Together they underpin `from_quadratic_surd`, `quadratic_coefficients`, and `quadratic_surd`.

5.5.1 Expanding a quadratic surd into a PSCF

The function `from_quadratic_surd_to_scf`(P, Q, D) computes the integer part, pre-period, and period of the PSCF whose value is $(P + \sqrt{D})/Q$. It proceeds in three stages: normalization,

iterative generation of complete quotients, and period detection via cycle recognition.

Stage 1: normalization. The algorithm requires the *divisibility invariant* $Q \mid (D - P^2)$ throughout its execution. To see why, note that each iteration produces a new denominator $d_{n+1} = (D - m_{n+1}^2)/d_n$: for this to be an integer, d_n must divide $D - m_{n+1}^2$. As the proof of Lemma 5.23 shows, this divisibility propagates automatically from step to step once it holds at $n = 0$, i.e. once $Q \mid (D - P^2)$. An arbitrary user-supplied triple need not satisfy this condition, so `normalize_quadratic_surd` rescales it first.

Specifically, the function finds the minimal positive integer

$$k = \frac{|Q|}{\gcd(|Q|,\, |D - P^2|)}$$

and replaces (P, Q, D) by $(kP,\, kQ,\, k^2D)$. The value is unchanged: $(kP + \sqrt{k^2D})/(kQ) = (P + \sqrt{D})/Q$. After rescaling, $kQ \mid k^2(D - P^2)$ holds because $|Q|$ divides $k \cdot |D - P^2| = \operatorname{lcm}(|Q|,\, |D - P^2|)$ by construction. The absolute value on Q is necessary to keep k positive (so that $\operatorname{sign}(Q)$ is preserved); the absolute value on $D - P^2$ ensures that the integer division $k = \operatorname{lcm}/|D - P^2|$ yields a positive result.

With the normalization in place we write (P, Q, D) for the rescaled parameters and record the established invariant:

$$Q \mid (D - P^2). \tag{N}$$

Stage 2: complete quotients. The standard strategy for computing the SCF of an irrational x is to iterate the map $x \mapsto 1/\{x\}$, where $\{x\} = x - \lfloor x \rfloor$ is the fractional part: the integer parts extracted along the way are precisely the partial quotients. Define

$$a_n = \lfloor x_n \rfloor, \qquad x_{n+1} = \frac{1}{x_n - a_n},$$

with $x_0 = (P + \sqrt{D})/Q$. The key structural observation is that, for quadratic surds, this iteration *preserves the surd form*: every complete quotient x_n is again of the form $(m_n + \sqrt{D})/d_n$ for integers m_n, d_n — with the *same* radicand D throughout. This is what makes the algorithm finite: the state (m_n, d_n) lives in a bounded integer lattice and must eventually repeat, forcing the SCF to be periodic.

Lemma 5.23 (Complete-quotient recurrence)**.** *Every complete quotient has the form $x_n = (m_n + \sqrt{D})/d_n$ where the pairs (m_n, d_n) satisfy the recurrence*

$$m_{n+1} = a_n\, d_n - m_n, \qquad d_{n+1} = \frac{D - m_{n+1}^2}{d_n}, \tag{R}$$

with base case $(m_0, d_0) = (P, Q)$. Moreover, the divisibility invariant $d_n \mid (D - m_n^2)$ holds for all $n \geq 0$, so every d_{n+1} is an integer; and $d_n \neq 0$ for all n, since $d_{n+1} = 0$ would require $m_{n+1}^2 = D$, contradicting the assumption that D is not a perfect square.

Proof. Recurrence. Suppose $x_n = (m_n + \sqrt{D})/d_n$. Define $m_{n+1} = a_n d_n - m_n$; then

$$x_n - a_n = \frac{\sqrt{D} - m_{n+1}}{d_n},$$

and inverting:

$$x_{n+1} = \frac{d_n}{\sqrt{D} - m_{n+1}} = \frac{d_n(\sqrt{D} + m_{n+1})}{D - m_{n+1}^2} = \frac{m_{n+1} + \sqrt{D}}{d_{n+1}}.$$

Divisibility invariant by induction. The base case is condition (N). Given $d_n \mid (D - m_n^2)$, expand

$$\begin{aligned} D - m_{n+2}^2 &= D - (a_{n+1}d_{n+1} - m_{n+1})^2 \\ &= (D - m_{n+1}^2) + 2a_{n+1}d_{n+1}m_{n+1} - a_{n+1}^2 d_{n+1}^2 \\ &= d_{n+1}(d_n + 2a_{n+1}m_{n+1} - a_{n+1}^2 d_{n+1}), \end{aligned}$$

using $D - m_{n+1}^2 = d_n d_{n+1}$. The right-hand side is divisible by d_{n+1}, so the invariant propagates. □ □

Stage 2 (continued): integer-arithmetic floor. The partial quotient $a_n = \lfloor (m_n + \sqrt{D})/d_n \rfloor$ must be computed exactly in pure integer arithmetic: floating-point evaluation of $\sqrt{D}$ can introduce rounding that corrupts the floor. The algorithm therefore uses $s = \lfloor \sqrt{D} \rfloor$, the integer square root computed exactly by `math.isqrt`, and evaluates `(m + s) // d` in Python, where `//` always rounds toward $-\infty$.

Case $d > 0$: no correction needed. Let $a = \lfloor (m+s)/d \rfloor$, so $ad \le m + s < (a+1)d$ (integer inequalities). We claim $a = \lfloor (m + \sqrt{D})/d \rfloor$:

- *Upper bound.* From $(a+1)d > m + s$ and integrality, $(a+1)d - m \ge s + 1$. Since $\sqrt{D}$ is irrational, $\sqrt{D} < s + 1$, so $(a+1)d - m > \sqrt{D}$, giving $(m + \sqrt{D})/d < a + 1$.
- *Lower bound.* $ad \le m + s$ gives $ad - m \le s < \sqrt{D}$, so $(m + \sqrt{D})/d > a$.

Combining, $a = \lfloor (m + \sqrt{D})/d \rfloor$ exactly.

Case $d < 0$: the candidate may overshoot by exactly one. Dividing by a negative number reverses the inequality $s < \sqrt{D}$, giving $(m+s)/d > (m + \sqrt{D})/d$. The two dividends differ by the positive amount $(\sqrt{D} - s)/|d|$; since $\sqrt{D} - s \in (0, 1)$, this difference lies in $(0, 1)$, so the floors of the two expressions can differ by at most 1: the candidate $a_{\text{cand}} = \lfloor (m+s)/d \rfloor$ may exceed the true floor a^* by at most 1.

To determine whether the overshoot occurs, define the *tentative next numerator* $m_1(a) = a \cdot d - m$, the value m_{n+1} would take if a were used as the partial quotient. The two candidates are separated by an if-and-only-if criterion:

$$a = a^* \quad \Longleftrightarrow \quad m_1(a) > \sqrt{D}. \tag{$\star$}$$

($\Rightarrow$) The floor definition with $d < 0$ reverses to $a^* d \ge m + \sqrt{D} > (a^* + 1)d$. The left inequality gives $m_1(a^*) \ge \sqrt{D}$; since $m_1(a^*)$ is an integer and $\sqrt{D}$ is irrational, $m_1(a^*) > \sqrt{D}$.

($\Leftarrow$) Since the candidate lies in $\{a^*, a^* + 1\}$, it suffices to show $m_1(a^* + 1) \le \sqrt{D}$. The right inequality $(a^* + 1)d < m + \sqrt{D}$ gives $m_1(a^* + 1) < \sqrt{D}$; as $m_1(a^* + 1)$ is an integer, $m_1(a^* + 1) \le s$. So $m_1(a) > \sqrt{D}$ rules out $a = a^* + 1$, forcing $a = a^*$.

A sharper observation: the overshoot condition $\lfloor (m+s)/d \rfloor = a^* + 1$ means (multiplying the floor inequality by $d < 0$) that $m_1(a^* + 1) \ge s$, and combined with $m_1(a^* + 1) \le s$ from above:

$$a_{\text{cand}} = a^* + 1 \iff m_1(a_{\text{cand}}) = s \quad \text{(exactly)}.$$

The correct case has $m_1 \ge s + 1 > s$; the overshoot case has $m_1 = s$. The code therefore checks `d < 0 and (a*d - m) <= s` and decrements by 1 if the condition fires; the $\le s$ form is the natural integer transcription of the strict irrational bound $m_1 < \sqrt{D}$.

After the first step, d is always positive. When $d_0 = Q < 0$, the correct a_0^* satisfies $m_1(a_0^*) > \sqrt{D}$, so $m_1^2 > D$, giving $d_1 = (D - m_1^2)/d_0 =$ (negative)/(negative) > 0. All subsequent denominators are positive, so the $d < 0$ branch is entered at most once over the entire execution.

Stage 3: period detection. Once $d_n > 0$ is established for all $n \geq 1$, we bound the state space of pairs (m_n, d_n).

Bounds on m_{n+1} and d_{n+1}. The fractional part $x_n - a_n \in (0, 1)$ gives, with $x_n = (m_n + \sqrt{D})/d_n$ and $d_n > 0$:

$$0 < \frac{\sqrt{D} - m_{n+1}}{d_n} < 1 \implies \sqrt{D} - d_n < m_{n+1} < \sqrt{D}.$$

The right inequality gives $m_{n+1} \leq s$ and $D - m_{n+1}^2 > 0$, confirming $d_{n+1} > 0$. For the upper bound on d_{n+1}, factor $D - m_{n+1}^2 = (\sqrt{D} - m_{n+1})(\sqrt{D} + m_{n+1})$; the left inequality gives $d_n > \sqrt{D} - m_{n+1} > 0$, so dividing,

$$d_{n+1} = \frac{(\sqrt{D} - m_{n+1})(\sqrt{D} + m_{n+1})}{d_n} < \sqrt{D} + m_{n+1} \leq 2\sqrt{D},$$

bounding d_{n+1} to at most $\lfloor 2\sqrt{D} \rfloor$ positive integer values. Applying this bound to d_n (inductively) and combining with the left inequality: $m_{n+1} > \sqrt{D} - d_n \geq \sqrt{D} - 2\sqrt{D} = -\sqrt{D}$, hence $m_{n+1} \geq -s$. Altogether $|m_{n+1}| \leq s$, confining m_{n+1} to at most $2s + 1$ integer values.

The state space of pairs (m_n, d_n) is therefore finite — at most $O(\sqrt{D}) \times O(\sqrt{D}) = O(D)$ states. By the pigeonhole principle the sequence must eventually revisit a state: there exist $n > k \geq 0$ with $(m_n, d_n) = (m_k, d_k)$, which implies $x_n = x_k$ and hence that all subsequent partial quotients repeat with period $n - k$.

The algorithm records each state's first occurrence in the `visited` dictionary. At the first revisit of (m_k, d_k), the accumulated list $[a_0, a_1, \ldots, a_{n-1}]$ is split as:

- Integer part: a_0;
- Pre-period: $(a_1, \ldots, a_{k-1})$ (empty when $k \leq 1$);
- Period: $(a_k, \ldots, a_{n-1})$.

The period is minimal because the loop halts at the *first* revisit: any shorter repeating block would imply an earlier recurrence of (m_k, d_k), contradicting the minimality of k.

Finally, note that $a_n \geq 1$ for all $n \geq 1$: since $x_{n+1} = 1/(x_n - a_n)$ and $x_n - a_n \in (0, 1)$, we have $x_{n+1} > 1$, so $a_{n+1} = \lfloor x_{n+1} \rfloor \geq 1$. All body partial quotients are therefore positive integers, as required by the SCF definition.

catena — `from_quadratic_surd_to_scf` and `from_quadratic_surd`

```
from catena.mathlib.convert import from_quadratic_surd_to_scf
from catena import PeriodicSimpleContinuedFraction as PSCF

# (1 + sqrt(5))/2  =  [1; 1,1,1,...]
from_quadratic_surd_to_scf(1, 2, 5)    # (1, (), (1,))

# Conjugate of phi: (-1 + sqrt(5))/(-2)  ~  -0.618
from_quadratic_surd_to_scf(-1, -2, 5)  # (-1, (), (1,))

# Eventually periodic: (7 + sqrt(10))/4
from_quadratic_surd_to_scf(7, 4, 10)   # (2, (1, 1), (5, 1, 1, 1,
   24, 1, 1, 1))

# from_quadratic_surd wraps the above and returns a PSCF directly
phi = PSCF.from_quadratic_surd(1, 2, 5)
phi.integer_part       # 1
phi.period             # (1,)
```

5.5.2 Recovering a quadratic surd from a PSCF

The inverse direction is implemented by `quadratic_coefficients()` and `quadratic_surd()`. The two steps are: derive the minimal polynomial from the periodic structure, then identify which root of that polynomial the PSCF represents.

Step 1: quadratic coefficients from the periodic structure. Consider first a PSCF with integer part zero and a purely periodic body $(c_0, c_1, \ldots, c_{r-1}, c_0, c_1, \ldots)$. Its value $y = [0; \overline{c_0, \ldots, c_{r-1}}]$ is a *fixed point* of the Möbius transformation obtained by composing the r elementary maps $t \mapsto c_j + 1/t$. Concretely, writing p_k/q_k for the k-th convergent of the finite SCF $[0; c_0, \ldots, c_{r-1}]$ (with the standard initialisation $p_{-1} = 1$, $p_0 = 0$, $q_{-1} = 0$, $q_0 = 1$), the standard evaluation formula gives

$$[0;\, c_0, \ldots, c_{r-1},\, t] = \frac{p_{r-1}\, t + p_{r-2}}{q_{r-1}\, t + q_{r-2}}.$$

Since $y = [0; c_0, \ldots, c_{r-1}, y]$ (the body repeats from position 0), setting $t = y$ and clearing denominators yields

$$q_{r-1}\, y^2 + (q_{r-2} - p_{r-1})\, y - p_{r-2} = 0.$$

The denominators q_k of convergents are strictly positive, so $A = q_{r-1} > 0$ holds without any sign correction; the $\operatorname{sign}(A)$ normalisation is included only for robustness.

For a PSCF with a non-empty pre-period $(d_0, \ldots, d_{m-1})$, the periodic tail $z = [0; \overline{c_0, \ldots, c_{r-1}}]$ satisfies the quadratic above, while the full body value $y = [0; d_0, \ldots, d_{m-1}, z]$ is a Möbius image of z through the pre-period convergents. Since Möbius transformations send quadratic irrationals to quadratic irrationals, y satisfies a (generally different) quadratic with integer coefficients.

`PeriodicGenerator.quadratic_coefficients()` computes the relevant convergents, forms $A = q_{r-1}$, $B = q_{r-2} - p_{r-1}$, $C = -p_{r-2}$ (for the purely periodic case), divides through by $\gcd(A, B, C)$ to obtain the primitive polynomial, and multiplies by $\operatorname{sign}(A)$ to enforce $A > 0$.

When the PSCF has integer part $a_0 \neq 0$, the full value $x = y + a_0$ satisfies the shifted polynomial with

$$A' = A, \quad B' = B - 2Aa_0, \quad C' = C - Ba_0 + Aa_0^2,$$

as derived in the remark following Definition 5.12.

Step 2: principal root extraction. Given the canonical triple (A, B, C) with $A > 0$, the principal root is

$$x_+ = \frac{-B + \sqrt{B^2 - 4AC}}{2A} = \frac{P + \sqrt{D}}{Q}, \qquad P = -B,\ Q = 2A,\ D = B^2 - 4AC.$$

Since $A > 0$ we have $Q > 0$, so x_+ is always the larger of the two roots.

Step 3: root disambiguation. Step 2 always produces the principal-root parameters (P, Q, D) with $Q > 0$, because the quadratic formula with $A > 0$ selects the $+\sqrt{D}$ branch. However, a given PSCF may represent the *conjugate* root. To decide which, `quadratic_surd()` calls `from_quadratic_surd_to_scf`(P, Q, D) to re-expand the principal root into its PSCF form and then compares the result with the stored integer part, pre-period, and period.

The comparison is sound because `from_quadratic_surd_to_scf` is deterministic: given (P, Q, D) it always produces the unique PSCF representation of the value $(P + \sqrt{D})/Q$. If the re-expansion matches the stored PSCF, the stored PSCF *is* the principal root and (P, Q, D) is returned unchanged. If it does not match, the stored PSCF is the conjugate root $(P - \sqrt{D})/Q = (-P + \sqrt{D})/(-Q)$, so $(-P, -Q, D)$ is returned, with $-Q < 0$ signalling the conjugate in accordance with Definition 5.15.

Remark 5.24 (Consistency of the round-trip)**.** The two algorithms are inverses of each other up to simplification:

- `from_quadratic_surd`(P, Q, D) followed by `quadratic_surd()` returns the simplified form (Definition 5.7) of (P, Q, D) — or its negation $(-P, -Q, D)$ if the input represented the conjugate root.
- `quadratic_surd()` followed by `from_quadratic_surd` reconstructs the original PSCF exactly.

In particular, `quadratic_surd()` always returns a simplified triple, while `from_quadratic_surd` accepts any triple and normalizes internally before running the expansion loop.

5.6 Inherited and additional methods

`PeriodicSimpleContinuedFraction` inherits all of `SimpleContinuedFraction` without change. In particular, `convergent`, `tail_convergent`, `tail`, `segment`, `shift`, and `__add__` all work on a PSCF exactly as described in the SCF section, since the underlying `PeriodicGenerator` is just another callable $g : \mathbb{N} \to \mathbb{Z}^+$. The convergent cache and the tail-sharing machinery are likewise inherited unchanged.

The following members are added or overridden by the class.

`non_repeating_part`, `period`
Read-only properties returning the pre-period $(g_{\text{pre}}(0), \ldots, g_{\text{pre}}(m-1))$ and the period $(g_{\text{per}}(0), \ldots, g_{\text{per}}(r-1))$ as immutable tuples. As it forces the entire pre-period and period to be materialized in memory as a new tuple, this may prove expensive for large periods; users should prefer the generator methods `pre_period_generator` and `period_generator` for large bodies.

`from_quadratic_surd(P, Q, D)`
Class method. Constructs a PSCF whose value is $(P+\sqrt{D})/Q$ by running the quadratic-surd expansion algorithm and forwarding the resulting pre-period, period, and integer part to the constructor. This is the factory used internally by `inverse` and `conjugate`.

`__float__` (override)
Evaluates the algebraic formula `(P + D**0.5) / Q` using the cached `quadratic_surd()` triple and Python's built-in float arithmetic for $\sqrt{D}$. In contrast, the base `SimpleContinuedFraction.__flo` approximates via the 50th convergent (a heuristic chosen for 64-bit float precision), and `FiniteSimpleContinuedFraction.__float__` uses the exact terminal convergent. The periodic override avoids convergent evaluation entirely by exploiting the closed-form surd.

`as_decimal()`
Returns the value as an arbitrary-precision `Decimal` by evaluating `(Decimal(P) + Decimal(D).sqrt()) / Decimal(Q)` from the cached `quadratic_surd()` triple. Precision is limited only by the active `decimal` context. The periodic analogue of `FiniteSimpleContinuedFraction.to_decimal()`, which evaluates the exact terminal convergent as a rational; here the equivalent exactness comes from the algebraic form.

`quadratic_coefficients()`
Returns (A, B, C) with $\gcd(A, B, C) = 1$ and $A > 0$ such that the value of the PSCF satisfies $Ax^2 + Bx + C = 0$.

`quadratic_surd()`
Returns (P, Q, D) such that the value equals $(P + \sqrt{D})/Q$, derived from the quadratic coefficients.

`is_principal_surd()`, `is_conjugate_root()`
: Predicates distinguishing the two roots of the minimal polynomial, identified by the sign of Q.

`inverse()` **(override)**
: Returns $1/x$ as a new PSCF. The result is a root of the *reflected polynomial* (Lemma 5.19); see Remark 5.21 for the four-object family. The write-once involution contract (`_inverse._inverse is self`) is preserved.

`conjugate()`
: Returns the algebraic conjugate $(P - \sqrt{D})/Q$, i.e. the other root of the same minimal polynomial.

`__neg__()`
: Returns $-x$ as a new PSCF via the surd negation $(P, -Q, D)$. The closed-form surd representation ensures that the definition of negation is exact, so no approximation is involved.

6 Generators

The basic notion of a *generator* — a function $g : I \to \mathbb{Z}^+$ encoding the body coefficients of an SCF — was introduced in Section **??** together with the three primitive types `Generator`, `CachedGenerator`, and `FiniteGenerator`. The present section focuses on the structural operations that transform one generator into another: `advance`, `insert`, and `prepend`. It also describes `PeriodicGenerator`, which extends the type hierarchy to eventually-periodic sequences.

6.1 Structural operations

All four generator types support three generator-to-generator operations. Their specifications are given uniformly here; the per-type notes below record any behavioural differences.

Definition 6.1 (Advance)**.** Let $g : \mathbb{N} \to \mathbb{Z}^+$ be a generator and $n \geq 0$ an integer. The *n-advance* of g is the generator

$$\mathrm{adv}_n(g)(k) \;=\; g(n+k), \qquad k \geq 0.$$

For $n = 0$, $\mathrm{adv}_0(g) = g$ (identity).

Remark 6.2. In terms of the SCF, advancing the generator by n discards the first n body coefficients: the resulting generator encodes the tail $[a_{n+1}; a_{n+2}, \ldots]$. Note that `SimpleContinuedFraction.tail()` (which takes no argument) is a special case: because the body generator already indexes from a_1, the tail $[0; a_1, a_2, \ldots]$ shares the *same* generator instance unchanged — only the integer part is set to zero. Advance is therefore not used by `tail` itself, but it is the primitive that would be needed to implement a hypothetical n-th tail operation.

Definition 6.3 (Insert)**.** Let $g : \mathbb{N} \to \mathbb{Z}^+$ and let $f = (b_0, \ldots, b_{s-1})$ be a `FiniteGenerator` of size s. For $\texttt{at} \geq 0$, the *insertion of f at position* `at` is the generator

$$\mathrm{ins}(g, f, \texttt{at})(k) = \begin{cases} g(k) & k < \texttt{at}, \\ b_{k-\texttt{at}} & \texttt{at} \leq k < \texttt{at} + s, \\ g(k-s) & k \geq \texttt{at} + s. \end{cases}$$

Definition 6.4 (Prepend)**.** The *prepend* of f onto g is $\mathrm{ins}(g, f, 0)$: the sequence $b_0, \ldots, b_{s-1}, g(0), g(1), \ldots$

Remark 6.5 (Prepend and the inverse operation)**.** Proposition 2.29 (case $a_0 > 0$) shows that inverting a positive SCF amounts to prepending the singleton $[a_0]$ to the body generator. Conversely, case $a_0 = 0$ removes the first body coefficient to form the new integer part. Both are special cases of advance/prepend on the generator.

catena — advance, insert, prepend

```
from catena.generators import Generator, FiniteGenerator

g = Generator(lambda n: n + 1)    # 1, 2, 3, 4, ...
f = FiniteGenerator([10, 20])

# advance: shift the index by 3 -> 4, 5, 6, ...
g3 = g.advance(3)
[g3(k) for k in range(4)]         # [4, 5, 6, 7]

# insert f at position 2: 1, 2, 10, 20, 3, 4, ...
h = g.insert(f, at=2)
[h(k) for k in range(6)]          # [1, 2, 10, 20, 3, 4]

# prepend: 10, 20, 1, 2, 3, ...
p = g.prepend(f)
```

```
[p(k) for k in range(5)]          # [10, 20, 1, 2, 3]
```

Type-specific behaviour.

`Generator`
: All three operations return a plain `Generator` wrapping a new closure; no memoisation is carried over.

`CachedGenerator`
: `advance` and `insert` accept an optional `copy_cache: bool` flag (default `False`). When `True`, cache entries are transferred to the new generator with indices adjusted to reflect the transformation: for an advance by n, entry (k, v) with $k \geq n$ is copied as $(k - n, v)$; for an insert at position p with block size s, entries before p are copied unchanged and entries from p onwards are shifted to $k + s$. When `False` the new generator starts with an empty cache, which is appropriate when the closure captured by the new generator differs from the original (e.g., it calls back into the original generator rather than the underlying function directly).

`FiniteGenerator`
: `advance`(n) returns the suffix $g[n:]$ as a new `FiniteGenerator`; advancing past the end raises `IndexError`. `insert` concatenates the underlying arrays, upgrading to the smallest unsigned typecode that covers all values in the combined sequence. There is no `prepend` override; it delegates to `insert(fg, at=0)`.

`PeriodicGenerator`
: See Section 6.2.

6.2 PeriodicGenerator

A `PeriodicGenerator` encodes an eventually-periodic sequence: a finite pre-period $(d_0, \ldots, d_{m-1})$ followed by an infinitely repeating period $(c_0, \ldots, c_{r-1})$:

$$g(k) = \begin{cases} d_k & 0 \leq k < m, \\ c_{(k-m) \bmod r} & k \geq m. \end{cases}$$

It is the generator type used internally by `PeriodicSimpleContinuedFraction`.

Advance. Let $n \geq 0$.

- If $n < m$: the new pre-period is $(d_n, \ldots, d_{m-1})$; the period is unchanged.
- If $n \geq m$: the pre-period is empty and the period is rotated by $(n - m) \bmod r$ positions:

$$(c_r, c_{r+1}, \ldots, c_{r-1}) \mapsto (c_{r'}, c_{r'+1}, \ldots, c_{r'-1}) \quad \text{where } r' = (n - m) \bmod r.$$

In both cases the result is a `PeriodicGenerator` with the same or shorter pre-period and a period that is a rotation of the original.

Insert. Inserting a `FiniteGenerator` f of size s at position `at`:

- If `at` $\leq m$: f is spliced into the pre-period; the period is unchanged.

- If `at` $> m$: one copy of the period is unrolled into the pre-period up to the insertion point, f is spliced in, and the original period is kept. Concretely, the new pre-period is

$$(d_0, \ldots, d_{m-1}, c_0, \ldots, c_{s'-1}, b_0, \ldots, b_{s-1}, c_{s'}, \ldots, c_{r-1}) \quad \text{where } s' = (\texttt{at} - m) \bmod r,$$

  and the period remains $(c_0, \ldots, c_{r-1})$.

Remark 6.6. After an insert with `at` $> m$, the period is *not* lengthened; only one instance of the period is unrolled into the pre-period to accommodate the inserted block. The resulting sequence is still eventually periodic with the same period, just a longer pre-period. This is the minimal representation: a shorter pre-period would require the inserted block to appear infinitely often, which is not the case.

catena — `PeriodicGenerator.advance` and `insert`

```
from catena.generators import FiniteGenerator, PeriodicGenerator

# period (1, 2, 3), pre-period (9, 8)
pg = PeriodicGenerator(period=[1, 2, 3], pre_period=[9, 8])
# sequence: 9, 8, | 1, 2, 3, 1, 2, 3, ...

# advance by 1: pre-period shrinks to (8,)
pg.advance(1).pre_period   # FiniteGenerator([8])
pg.advance(1).period       # FiniteGenerator([1, 2, 3])

# advance by 3 (past the pre-period): period rotates by (3-2)%3 = 1
pg.advance(3).pre_period   # FiniteGenerator([])
pg.advance(3).period       # FiniteGenerator([2, 3, 1])

# insert [7] at position 4 (within the first copy of the period):
# pre-period becomes (9, 8, 1, 2, 7, 3), period stays (1, 2, 3)
ins = pg.insert(FiniteGenerator([7]), at=4)
ins.pre_period             # FiniteGenerator([9, 8, 1, 2, 7, 3])
ins.period                 # FiniteGenerator([1, 2, 3])
[ins(k) for k in range(10)]
# [9, 8, 1, 2, 7, 3, 1, 2, 3, 1]
```

Quadratic coefficients. `PeriodicGenerator` also exposes `quadratic_coefficients()`, which returns the primitive polynomial (A, B, C) satisfied by the value of the generator (with integer part zero). The derivation is described in Section 5.5.2; it is not repeated here.

7 The `catena.numbers` module

The `catena.numbers` module provides two sub-modules: `constants`, which pre-constructs SCF representations of well-known mathematical constants, and `randoms`, which supplies generators and SCF factories whose coefficients are drawn from the Gauss–Kuzmin distribution. The present section describes both. The mathematical justification for the random generators requires results on the measure theory of continued fractions that go beyond the scope of this version of the document; the relevant facts are stated without proof and deferred to a forthcoming section.

7.1 Constants

The module exposes the following pre-built SCF objects.

`e` Euler's number as a `SimpleContinuedFraction` with integer part 2 and body generator

$$g(n) = \begin{cases} 2(\lfloor (n+1)/3 \rfloor + 1) & n \equiv 2 \pmod 3, \\ 1 & \text{otherwise.} \end{cases}$$

This encodes the classical pattern $e = [2;\, 1, 2, 1,\ 1, 4, 1,\ 1, 6, 1, \ldots]$, where every third body coefficient (at positions $n = 2, 5, 8, \ldots$) equals $2, 4, 6, \ldots$ and the remaining coefficients are 1. The generator is stateless and computed in $O(1)$.

`phi`
The golden ratio $\varphi = [1;\, \overline{1}]$ as a `PeriodicSimpleContinuedFraction` with integer part 1 and period (1).

`sqrt2`, `sqrt3`, `sqrt5`
The PSCFs $[1;\, \overline{2}]$, $[1;\, \overline{1, 2}]$, $[2;\, \overline{4}]$ for $\sqrt{2}$, $\sqrt{3}$, $\sqrt{5}$ respectively.

`metallic_mean`(n)
Factory returning the n-th metallic mean $(n + \sqrt{n^2 + 4})/2 = [n;\, \overline{n}]$ as a PSCF. For $n = 1$ this recovers φ; for $n = 2$ the silver mean, etc.

catena — `catena.numbers.constants`

```
from catena.numbers.constants import e, phi, sqrt2, sqrt3, sqrt5,
    metallic_mean

e.integer_part              # 2
e.generator(0)              # 1  (first body coefficient)
e.generator(2)              # 2  (a_3 = 2)
e.generator(5)              # 4  (a_6 = 4)

phi.period                  # (1,)
sqrt2.period                # (2,)
sqrt3.period                # (1, 2)
metallic_mean(3).period     # (3,)   -- bronze mean = [3; 3, 3, ...]
```

7.2 Random SCFs and the Gauss–Kuzmin distribution

Mathematical background (stated without proof). The theory of continued fractions has a natural probabilistic counterpart. Equip $[0, 1)$ with the Gauss measure μ, whose density with respect to Lebesgue measure is

$$\frac{d\mu}{d\lambda}(x) = \frac{1}{\ln 2} \cdot \frac{1}{1 + x}.$$

Under μ, the partial quotients $a_1, a_2, \ldots$ of a *random* continued fraction (drawn uniformly from $[0,1)$ with respect to μ) are independent and identically distributed with the *Gauss–Kuzmin distribution*:

$$\Pr[a_n = k] = \log_2\left(1 + \frac{1}{k(k+2)}\right), \qquad k \geq 1. \tag{GK}$$

This result, due to Gauss (observed) and Kuzmin (proved), is the foundational theorem of the ergodic theory of continued fractions. A full proof, and the connection to the Gauss–Kuzmin–Wirsing operator, is deferred to a forthcoming section.

The Gauss–Kuzmin distribution can be sampled via the inverse-transform method: if $U \sim \text{Uniform}(0,1)$ then

$$K = \left\lfloor \frac{1}{2^U - 1} \right\rfloor \tag{IT}$$

has the Gauss–Kuzmin distribution. The library uses this formula directly.

Implementation. catena provides two concrete samplers and one abstract base class.

`Seed`
: A stateful SHA-256 chain: each access to `.state` returns the current bytes and advances the internal state by one hash round. The default global `_SEED` is initialised from a random UUID, so different process invocations produce independent sequences.

`GaussKuzminSHA`
: Stateless sampler for a fixed seed. For index n, it derives a 53-bit uniform integer via SHA-256 of $(seed \| n)$, converts it to a float $U \in [0,1)$, and applies (IT). Each call is $O(1)$ and the outputs for different n are independent (under the cryptographic assumption that SHA-256 behaves as a random oracle).

`GaussKuzminSHAArbitrary`
: As `GaussKuzminSHA` but the uniform sample U is computed to an arbitrary number of significant decimal digits using `UniformSHAArbitrary` (a SHA-256 chain that assembles as many 256-bit blocks as needed). The result of (IT) is then an exact Python `int`, avoiding any float truncation that could bias the floor computation for large partial quotients.

`RandomSCF` (abstract base class)
: Defines the factory interface: `generator()`, `cached_generator()`, `finite_generator(size)`, `periodic_generator(period_size, pre_period_size)`, `scf()`, `finite_scf(size)`, `periodic_scf(...`. Concrete subclasses supply `__make_callable__()`, which returns a fresh callable backed by a new SHA seed drawn from the instance's `Seed`.

`GaussKuzminSCF`, `GaussKuzminArbitrarySCF`
: Concrete subclasses of `RandomSCF` backed by `GaussKuzminSHA` and `GaussKuzminSHAArbitrary` respectively.

catena — catena.numbers.randoms

```
from catena.numbers.randoms import GaussKuzminSCF,
    GaussKuzminArbitrarySCF, Seed

# reproducible: fix the seed
rng = GaussKuzminSCF(seed=Seed(42))
scf = rng.scf(integer_part=0)         # infinite SCF with GK-
    distributed body
scf.convergent(10)                     # (p_10, q_10) -- some large
    rational

```

```
# finite sample of length 20
fscf = rng.finite_scf(size=20)       # FiniteSimpleContinuedFraction

# arbitrary-precision sampler
rng2 = GaussKuzminArbitrarySCF(precision=100, seed=Seed("hello"))
g = rng2.generator()
g(0)                                 # a positive integer drawn from
    GK

# periodic SCF with 5-term period, no pre-period
pscf = rng.periodic_scf(period_size=5, integer_part=3)
pscf.period                          # tuple of 5 positive integers
```

Remark 7.1 (Determinism and reproducibility)**.** Because each call to `__make_callable__()` draws a fresh SHA seed from the instance's `Seed`, two successive calls to `rng.generator()` produce *different* (independent) generators. A single `Seed` object therefore acts as a splittable PRNG: the sequence of seeds it emits is deterministic given the initial state, and each emitted seed drives a fully independent coefficient sequence. Passing `seed=Seed(k)` with a fixed integer k makes the entire computation reproducible.

Remark 7.2 (Forthcoming: measure theory and distribution of convergents)**.** The Gauss–Kuzmin law (GK) is the marginal distribution of any single partial quotient of a μ-random continued fraction. Several deeper results — the exponential mixing of the Gauss map, the Gauss–Kuzmin–Wirsing spectral gap, Khintchine's theorem on the geometric mean of partial quotients, and the distribution of convergent denominators — are not yet covered in this document and are planned for a forthcoming section on the ergodic theory of continued fractions.

8 Planned Extensions

The following are flagged as future development areas; they are not yet implemented in the library, but they are on the roadmap. That said, no promises are made about the timeline for any of these features, and they may be subject to change as the project evolves. Each will warrant its own section in this document.

8.1 Arithmetic on Continued Fractions

Gosper (1972) showed how to perform the four arithmetic operations on SCFs in a streaming fashion using 2×2 and $2 \times 2 \times 2$ integer matrices, without fully evaluating either operand. This is the natural future direction for `FiniteSimpleContinuedFraction` arithmetic. See [1]

8.2 Generalised Continued Fractions

The general form $b_0 + K_{n=1}^{\infty} a_n / b_n$ with arbitrary integer or function sequences. Applications: rapid evaluation of π, e, $\ln 2$, Bessel functions, etc.

8.3 Metric Theory and Gauss Map

The map $T : x \mapsto \{1/x\}$ (fractional part of $1/x$) on $(0, 1)$ is the *Gauss map*; its invariant measure $\mu(A) = \frac{1}{\ln 2} \int_A \frac{dx}{1+x}$ governs the distribution of partial quotients (Gauss–Kuzmin statistics). This will be relevant for statistical analysis routines.

8.4 Multidimensional Generalisations

Jacobi–Perron and related algorithms for simultaneous approximation; connection to lattice reduction (LLL algorithm).

9 Notation Index

Symbol	catena attribute	Meaning
$[a_0;\, a_1, \ldots, a_n]$	—	Finite simple continued fraction
a_0	`integer_part`	Integer part of the SCF
$g : I \to \mathbb{Z}^+$	`generator`	Callable partial-quotient generator
$(a_1, \ldots, a_k)$	`partial_quotients`	Body sequence (finite case only)
$k = \lvert I \rvert$	`size`	Number of body elements (finite case)
$x_n = (a_0, g\lvert_n)$	`segment(`n`)`	n-th segment (truncation to first n body quotients)
$t(x) = (0, g)$	`tail()`	Tail: SCF with integer part set to zero
(h_n, k_n)	`tail_convergent(`n`)`	n-th tail convergent numerator/denominator
(p_n, q_n)	`convergent(`n`)`	n-th full convergent numerator/denominator
$r_n(x) = (g(n), h)$	—	n-th remainder (not yet implemented in catena)
$\{\cdot\}$	—	Fractional part $\{x\} = x - \lfloor x \rfloor$
$f\lvert_S$	—	Restriction of f to subset S of its domain
T	—	Gauss map $x \mapsto \{1/x\}$

Remark 9.1. The notation table entry `body` from earlier drafts has been replaced. In the codebase, `body` (or `partial_quotients`) refers to the finite *sequence* $(a_1, \ldots, a_k)$, while `generator` is the callable $g : I \to \mathbb{Z}^+$. These are distinct: the generator is an arbitrary function, whereas the partial-quotients sequence is its finite materialisation. The paper uses g throughout; `partial_quotients` is the `FiniteSimpleContinuedFraction`-specific attribute that stores the precomputed values.

References

[1] R. William Gosper. *Continued Fraction Arithmetic*. Tech. rep. HAKMEM Item 101B. Unpublished memo. MIT Artificial Intelligence Laboratory, 1972.

[2] William B. Jones and Wolfgang J. Thron. *Continued Fractions: Analytic Theory and Applications*. Vol. 11. Encyclopedia of Mathematics and Its Applications. Addison-Wesley, 1980.

[3] Aleksandr Yakovlevich Khinchin. *Continued Fractions*. English translation of the 3rd Russian edition (1961). Dover reprint 1997. University of Chicago Press, 1964.

[4] Karen Lynn Parrish. "A Study in Applications of Continued Fractions". CSUSB ScholarWorks, Electronic Theses, Projects, and Dissertations, No. 1371. California State University, San Bernardino, Dec. 2021. URL: `https://scholarworks.lib.csusb.edu/etd/1371`.

[5] Andrew M. Rockett and Peter Szüsz. *Continued Fractions*. World Scientific, 1992.