q-Kangaroo Reference Manual

Symbolic q-Series Computation

Version `version`

A comprehensive reference for all 81 built-in functions,
the expression language, CLI usage, and worked examples.

q-Kangaroo Contributors

Contents

1 Quick Start

This chapter walks you through installing q-Kangaroo, evaluating your first q-series expression, and writing a short script. For detailed installation instructions, see Section 2.

1.1 Installing q-Kangaroo

Download the pre-built binary for your platform from the GitHub Releases page. On Linux or macOS, extract the archive and place `q-kangaroo` somewhere on your `PATH`. On Windows, extract the zip and optionally add the directory to your `PATH`. See Chapter 2 for full details.

Verify the installation:

```
q> --version
q-kangaroo 0.9.0
```

1.2 Your First Expression

Launch the interactive REPL by running `q-kangaroo` with no arguments. You will see the ASCII kangaroo banner and a `q>` prompt.

Start with the partition generating function. The coefficient of q^n in `partition_gf(N)` is $p(n)$, the number of integer partitions of n:

```
q> partition_gf(10)
1 + q + 2*q^2 + 3*q^3 + 5*q^4 + 7*q^5 + 11*q^6 + 15*q^7 + 22*q^8 + 30*q^9 + O(q^10)
```

The `O(q^10)` term indicates that the series is truncated at order 10. To compute the exact number of partitions of 100 as an integer:

```
q> partition_count(100)
190569292
```

1.3 Exploring q-Series

Assign a q-Pochhammer product to a variable using `:=`:

```
q> f := aqprod(1, 1, 1, infinity, 20):
q> f
1 - q - q^2 + q^5 + q^7 - q^12 - q^15 + O(q^20)
```

The `:` at the end of the first line suppresses output. The variable `f` now holds $(q;q)_\infty$ truncated to order 20.

Use `%` to reference the last result. Here we analyze the product form of the series:

```
q> prodmake(%, 10)
{1: 1, 2: 1, 3: 1, 4: 1, 5: 1, 6: 1, 7: 1, 8: 1, 9: 1, 10: 1}
```

This confirms that $f = \prod_{k=1}^{\infty}(1-q^k)$, with exponent 1 for each factor $(1-q^k)$.

1.4 Your First Script

Create a file called `example.qk` with the following contents:

```
# example.qk -- Euler's pentagonal number theorem
f := aqprod(1, 1, 1, infinity, 20):
g := partition_gf(20):
f * g
```

Run the script from the command line:

```
$ q-kangaroo example.qk
1 + O(q^20)
```

The result `1` confirms the identity $(q;q)_\infty \cdot \frac{1}{(q;q)_\infty} = 1$.

1.5 Getting Help

At the `q>` prompt, type `help` to see a list of all 81 built-in functions grouped by category. To see detailed help for a specific function:

```
q> help aqprod
aqprod(coeff_num, coeff_den, power, n_or_infinity, order)

  Compute the q-Pochhammer product (a;q)_n where a = (coeff_num/coeff_den)*q^power.
  When n is 'infinity', computes the infinite product (a;q)_inf.

  Example:
    q> aqprod(1, 1, 1, infinity, 10)
    1 - q - q^2 + q^5 + q^7 + O(q^10)
```

From the command line, `q-kangaroo --help` shows available flags and usage.

2 Installation

q-Kangaroo is distributed as a single static binary with no runtime dependencies. Pre-built binaries are available for Linux (x86_64) and Windows (x86_64). A Python API package is also available.

2.1 Pre-built Binaries

Download the latest release from the GitHub Releases page. Each release includes:

- `q-kangaroo-linux-x86_64.tar.gz` – Linux binary
- `q-kangaroo-windows-x86_64.zip` – Windows binary
- `q-kangaroo-manual.pdf` – This reference manual

2.2 Linux

```
tar xzf q-kangaroo-linux-x86_64.tar.gz
chmod +x q-kangaroo
sudo mv q-kangaroo /usr/local/bin/
```

Alternatively, place the binary anywhere on your `PATH`.

2.3 Windows

Extract the zip archive. The `q-kangaroo.exe` binary can be run directly from any directory. Optionally, add the directory containing the executable to your system `PATH` for convenient access from any command prompt.

2.4 Building from Source

Building q-Kangaroo from source requires:

- **Rust 1.85 or later** (install via rustup)
- **C compiler** (for GMP/MPFR/MPC, which are compiled from source automatically)

Clone the repository and build the release binary:

```
git clone https://github.com/q-kangaroo/q-kangaroo.git
cd q-kangaroo
cargo build --release -p qsym-cli
```

The binary is produced at `target/release/q-kangaroo` (or `q-kangaroo.exe` on Windows). The first build takes several minutes because GMP, MPFR, and MPC are compiled from source. Subsequent builds are fast due to caching.

2.5 Python API

The Python bindings are available as a separate package:

```
pip install q-kangaroo
```

The Python API provides the same 81 functions through a `QSession` object. See the Python documentation for details.

2.6 Verifying Installation

Run `q-kangaroo --version` to confirm the binary is installed correctly:

```
q> --version
q-kangaroo 0.9.0
```

If the REPL launches but you see unexpected behavior, try `q-kangaroo -v` (verbose mode) to see per-statement timing, which can help diagnose issues.

3 Command-Line Interface

q-Kangaroo supports four execution modes: interactive REPL, script execution, expression evaluation, and piped input. This chapter documents all command-line flags, execution modes, session commands, and exit codes.

3.1 Synopsis

```
q-kangaroo [OPTIONS] [FILE]
q-kangaroo -c EXPRESSION
command | q-kangaroo
```

3.2 Options

Flag	Description
`-h`, `--help`	Show usage information and exit.
`-V`, `--version`	Print version string and exit.
`-c EXPRESSION`	Evaluate the given expression, print the result, and exit. The expression argument is required.
`-q`, `--quiet`	Suppress the ASCII banner in interactive mode. Has no effect in other modes.
`-v`, `--verbose`	Show per-statement execution timing on stderr. Works in all modes.
`--`	End of options. The next positional argument is treated as a filename, even if it starts with `-`.

3.3 Execution Modes

q-Kangaroo automatically selects its execution mode based on how it is invoked:

3.3.1 Interactive REPL

When invoked with no file argument and stdin is a terminal, q-Kangaroo enters the interactive Read-Eval-Print Loop. Features include:

- Line editing with Emacs keybindings (via rustyline)
- Persistent command history (stored next to the executable)
- Tab completion for function names and user variables
- Multi-line input (open parentheses continue to the next line)
- Session commands (see below)

```
$ q-kangaroo
                                    /)
                                  '  \
       ... (ASCII kangaroo banner) ...
```

```
Type 'help' for commands
'quit' to exit

q> partition_gf(5)
1 + q + 2*q^2 + 3*q^3 + 5*q^4 + O(q^5)
```

3.3.2 Script Execution

Pass a filename as a positional argument to execute a script:

```
$ q-kangaroo script.qk
```

Scripts use the same expression language as the REPL. Lines starting with `#` are comments. The script stops on the first error (fail-fast semantics). Error messages include `filename:line:` context.

3.3.3 Expression Evaluation

The `-c` flag evaluates a single expression:

```
$ q-kangaroo -c "etaq(1, 1, 20)"
q^(1/24) * (1 - q - q^2 + q^5 + q^7 + ... + O(q^20))
```

Multiple statements can be separated by `;` within the expression string.

3.3.4 Piped Input

When stdin is not a terminal, q-Kangaroo reads all input and evaluates it as a script:

```
$ echo "1 + 1" | q-kangaroo
2
```

3.4 Session Commands

In interactive mode, the following commands are recognized before the expression parser. They are case-insensitive.

Command	Description
`help`	Display all 81 functions grouped by category, plus session commands.
`help` *function*	Show detailed help for a specific function: signature, description, and example.
`set precision N`	Set the default truncation order for series computation. The default is 20. The value must be a positive integer.
`clear`	Reset all user variables, the last result (`%`), and the truncation order back to 20.
`quit` / `exit`	Exit the REPL. Ctrl-D (EOF) also exits. Ctrl-C cancels the current line without exiting.
`latex` *[var]*	Display the LaTeX representation of the last result, or of a named variable.

Command	Description
`save` *filename*	Save the text representation of the last result to a file.
`read` *filename*	Load and execute a script file within the current session. Variables defined in the script become available at the REPL prompt. Alternatively, use the function form `read("filename.qk")` in expressions.

3.5 Exit Codes

q-Kangaroo uses sysexits-compatible exit codes for scripting integration:

Code	Name	Meaning
`0`	Success	All statements executed successfully.
`1`	Eval error	A runtime evaluation error occurred (e.g., undefined variable, type mismatch, division by zero).
`2`	Usage error	Invalid command-line arguments (unknown flag, missing `-c` argument).
`65`	Parse error	Syntax error in input (unmatched parenthesis, invalid token, etc.).
`66`	File not found	Script file does not exist or is unreadable.
`70`	Panic	An internal computation error was caught (e.g., overflow in the core library).
`74`	I/O error	File I/O failure other than "not found" (permission denied, disk full, etc.).

In interactive mode, errors are displayed and the REPL continues. In script and expression modes, execution stops on the first error and the appropriate exit code is returned.

3.6 Error Messages

In script and expression modes, parse errors include source location context:

```
script.qk:3:5: unexpected token ')'
```

Evaluation errors in scripts include the filename and line number:

```
script.qk:7: undefined variable 'x'
```

In interactive mode, errors are printed without filename context and the REPL continues accepting input. Panics from the core library are caught and translated to user-friendly messages.

4 Expression Language

q-Kangaroo uses a Maple-inspired expression language for all input, whether typed at the interactive prompt, written in a script file, or passed via the `-c` flag. This chapter describes the complete language syntax.

4.1 Overview

The expression language supports integer and rational arithmetic, formal power series in the indeterminate q, variable assignment, function calls, lists, and two statement terminators that control output. There are no control-flow statements (loops, conditionals) – the language is designed for evaluating mathematical expressions, not general-purpose programming.

4.2 Literals

4.2.1 Integer Literals

Integer literals are written in decimal notation. They have arbitrary precision – there is no upper limit on the number of digits:

```
q> 99999999999999999999999999 + 1
100000000000000000000000000
```

4.2.2 The `q` Indeterminate

The symbol `q` is a reserved keyword representing the formal indeterminate of power series. It cannot be used as a variable name:

```
q> q
q
```

```
q> q^2 + q + 1
1 + q + q^2
```

4.2.3 The `infinity` Keyword

The keyword `infinity` is used as an argument to functions that accept either a finite bound or an infinite product:

```
q> aqprod(1, 1, 1, infinity, 10)
1 - q - q^2 + q^5 + q^7 + O(q^10)
```

4.2.4 String Literals

Double-quoted strings are used for filenames in the `read()` function and the `save` command:

```
q> read("script.qk")
```

Supported escape sequences: `\\` (backslash), `\"` (double quote), `\n` (newline), `\t` (tab).

4.3 Variables and Assignment

Variables are bound using the `:=` assignment operator:

```
q> f := partition_gf(20):
q> g := distinct_parts_gf(20):
q> f
1 + q + 2*q^2 + 3*q^3 + 5*q^4 + ... + O(q^20)
```

Variable names consist of letters, digits, and underscores, and must start with a letter. Names are case-sensitive: `f` and `F` are different variables.

4.3.1 Last Result Reference (`%`)

The special symbol `%` refers to the value of the most recently printed result:

```
q> partition_gf(10)
1 + q + 2*q^2 + 3*q^3 + 5*q^4 + 7*q^5 + 11*q^6 + 15*q^7 + 22*q^8 + 30*q^9 + O(q^10)
q> prodmake(%, 5)
{1: -1, 2: -1, 3: -1, 4: -1, 5: -1}
```

4.4 Arithmetic Operators

Operator	Syntax	Description
`+`	$a + b$	Addition. Works on integers, rationals, and series.
`-`	$a - b$	Subtraction. Also unary negation: $-a$.
`*`	$a * b$	Multiplication. Series are multiplied with truncation.
`/`	a/b	Division. Integer division produces a rational. Series division uses power series inversion.
`^`	$a \hat{} b$	Exponentiation. For series, the exponent must be an integer. For integers, both base and exponent must be non-negative.

Operator precedence follows standard mathematical conventions: `^` binds tightest, then `*` and `/`, then `+` and `-`. Parentheses override precedence as usual.

```
q> 2 + 3 * 4
14
```

```
q> (2 + 3) * 4
20
```

4.5 Lists

Lists are written with square brackets and comma-separated elements:

```
q> [1, 2, 3]
[1, 2, 3]
```

Lists are used as arguments to several functions. For example, `findlincombo` takes a list of candidate series, and `phi` / `psi` take lists of parameter triples for the upper and lower parameters of hypergeometric series:

```
q> findlincombo(partition_gf(30), [distinct_parts_gf(30), odd_parts_gf(30)], 0)
[0, 1]
```

4.6 Function Calls

Functions are called with the standard `name(arg1, arg2, ...)` syntax. q-Kangaroo provides 81 built-in functions organized into 8 groups:

- **Products** (7): `aqprod`, `qbin`, `etaq`, `jacprod`, `tripleprod`, `quinprod`, `winquist`
- **Partitions** (7): `partition_count`, `partition_gf`, `distinct_parts_gf`, `odd_parts_gf`, `bounded_parts_gf`, `rank_gf`, `crank_gf`
- **Theta Functions** (3): `theta2`, `theta3`, `theta4`
- **Series Analysis** (9): `sift`, `qdegree`, `lqdegree`, `qfactor`, `prodmake`, `etamake`, `jacprodmake`, `mprodmake`, `qetamake`
- **Relations** (12): `findlincombo`, `findhomcombo`, `findnonhomcombo`, and 9 others
- **Hypergeometric** (9): `phi`, `psi`, `try_summation`, `heine1`–`heine3`, `sears_transform`, `watson_transform`, `find_transformation_chain`
- **Mock Theta & Bailey** (27): 20 mock theta functions, 3 Appell-Lerch/universal, 4 Bailey chain
- **Identity Proving** (7): `prove_eta_id`, `search_identities`, `q_gosper`, `q_zeilberger`, `verify_wz`, `q_petkovsek`, `prove_nonterminating`

See Chapters 5–12 for complete documentation of every function.

4.7 Statement Separators

Multiple statements can appear on one line, separated by `;` or `:`:

- **Semicolon** (`;`): Evaluate and **print** the result.
- **Colon** (`:`): Evaluate and **suppress** the output.
- **End of line**: Implicitly prints the result (same as `;`).

```
q> a := 2; b := 3; a + b
2
3
5
```

```
q> a := 2: b := 3: a + b
5
```

In the second example, the assignments are suppressed by `:`, so only the final expression `a + b` produces output.

4.8 Comments

In scripts, lines starting with `#` are comments:

```
# This is a comment
f := etaq(1, 1, 20)  # inline comments also work
f
```

Comments extend from `#` to the end of the line.

4.9 Value Types

Every expression in q-Kangaroo evaluates to one of the following types:

Type	Description
Series	A formal power series in q with rational coefficients, truncated at a specified order. Produced by most q-series functions (`aqprod`, `etaq`, `partition_gf`, `theta3`, etc.).
Integer	An arbitrary-precision integer. Produced by `partition_count`, `qdegree`, `lqdegree`, and integer arithmetic.
Rational	An exact rational number p/q. Produced by integer division and coefficient extraction.
List	An ordered collection of values. Produced by `[a, b, c]` syntax and functions like `findcong`.
Dict	A key-value mapping. Produced by `prodmake`, `etamake`, `qfactor`, `jacprodmake`, `mprodmake`, and `qetamake`.
Pair	A pair of two values. Produced by Heine transformations and Bailey lemma functions (which return `(prefactor, transformed_series)` or `(alpha, beta)` pairs).
Bool	A boolean value (`true` or `false`). Produced by `prove_eta_id` and `verify_wz`.
String	A text string. Used for filenames in `read()` and `save`.
None	The null value. Returned by `try_summation` when no closed form is found, and by `bailey_discover` when no proof is found.
Infinity	The `infinity` keyword. Used as a parameter to `aqprod` to request an infinite product.

Arithmetic operations automatically promote types where sensible: adding an integer to a series produces a series, dividing two integers produces a rational, and so on.

5 Products

Infinite products lie at the heart of q-series theory. The q-Pochhammer symbol $(a;q)_n$ is the fundamental building block from which nearly every other q-series object – Dedekind eta functions, Jacobi theta functions, partition generating functions, and the Rogers–Ramanujan products – is constructed. q-Kangaroo provides seven product functions covering the q-Pochhammer symbol, q -binomial coefficients, Dedekind eta quotients, and several classical named products.

All product functions return truncated formal power series in q. The `order` parameter controls the truncation: terms of degree $\geq$ order are discarded and represented as $O\left(q^{\text{order}}\right)$.

5.1 Function Reference

5.1.1 aqprod

```
aqprod(coeff_num, coeff_den, power, n_or_infinity, order)
```

Compute the q-Pochhammer product $(a;q)_n$ where $a = (\text{coeff\_num} \,/\, \text{coeff\_den}) \cdot q^{\text{power}}$. When `n_or_infinity` is the keyword `infinity`, computes the infinite product $(a;q)_\infty$. This is the most fundamental building block in q-series theory: every other product function in q-Kangaroo is ultimately expressed in terms of `aqprod`.

Mathematical Definition

$$(a;q)_n = \prod_{k=0}^{n-1} \left(1 - aq^k\right)$$

For the infinite product:

$$(a;q)_\infty = \prod_{k=0}^{\infty} \left(1 - aq^k\right)$$

With the parametrization $a = (c_1/c_2)q^p$, the call `aqprod(c1, c2, p, n, order)` computes $\left(c_1/c_2 \cdot q^p;q\right)_n$ truncated to $O\left(q^{\text{order}}\right)$.

Parameters

Name	Type	Description
coeff_num	Integer	Numerator of the coefficient a
coeff_den	Integer	Denominator of the coefficient a (must be nonzero)
power	Integer	Power of q in the base: $a = (c_1/c_2)q^{\text{power}}$
n_or_infinity	Integer or `infinity`	Number of factors, or `infinity` for the infinite product
order	Integer	Truncation order for the result

Examples

```
q> aqprod(1, 1, 1, infinity, 10)
1 - q - q^2 + q^5 + q^7 + O(q^10)
```

```
q> aqprod(1, 1, 1, 3, 20)
1 - q - q^2 + q^4 + q^5 - q^6 + O(q^20)
```

```
q> aqprod(1, 2, 1, infinity, 10)
1 - 1/2*q - 1/2*q^2 - 1/4*q^3 - 1/4*q^4 - 1/8*q^6 + 1/8*q^7 + 1/8*q^9 + O(q^10)
```

Edge Cases and Constraints

- `coeff_den` must be nonzero; division by zero produces an error.
- `n_or_infinity` must be a non-negative integer or the keyword `infinity`.
- `order` must be a positive integer.
- The Euler function $(q;q)_\infty$ is obtained by `aqprod(1, 1, 1, infinity, order)`.

Related: etaq, tripleprod, jacprod, quinprod

5.1.2 qbin

```
qbin(n, k, order)
```

Compute the q-binomial coefficient (Gaussian binomial coefficient) $\binom{n}{k}_q$. The result is a polynomial in q of degree $k(n-k)$. The q-binomial coefficient is the q-analog of the ordinary binomial coefficient and arises as the generating function for partitions into at most k parts, each at most $n-k$.

Mathematical Definition

$$\binom{n}{k}_q = \frac{(q;q)_n}{(q;q)_k (q;q)_{n-k}}$$

This is a polynomial in q of degree $k(n-k)$ with non-negative integer coefficients. At $q=1$ it reduces to the ordinary binomial coefficient $\binom{n}{k}$.

Parameters

Name	Type	Description
n	Integer	Upper index (must be non-negative)
k	Integer	Lower index (must satisfy $0 \le k \le n$)
order	Integer	Truncation order for the result

Examples

```
q> qbin(4, 2, 20)
1 + q + 2*q^2 + q^3 + q^4 + O(q^20)
```

```
q> qbin(5, 0, 20)
1 + O(q^20)
```

```
q> qbin(5, 3, 20)
1 + q + 2*q^2 + 2*q^3 + 2*q^4 + q^5 + q^6 + O(q^20)
```

Edge Cases and Constraints

- k must satisfy $0 \leq k \leq n$; otherwise an error is produced.
- The result is a polynomial of degree $k(n-k)$, so terms beyond that degree are zero.
- `qbin(n, 0, order)` and `qbin(n, n, order)` both return 1.

Related: aqprod, partition_gf, bounded_parts_gf

5.1.3 etaq

```
etaq(b, t, order)
```

Compute the generalized Dedekind eta quotient. The parameter b is the base (controlling the spacing of factors in the product) and t is the exponent. This function computes the product form of the Dedekind eta function raised to the power t, including the leading q-shift $q^{bt/24}$.

Mathematical Definition

$$\eta_b^t = q^{bt/24} \prod_{k=1}^{\infty} \left(1 - q^{bk}\right)^t$$

The classical Dedekind eta function is $\eta(\tau) = q^{1/24} \prod_{k=1}^{\infty}(1-q^k)$, obtained by `etaq(1, 1, order)`. Setting $t = -1$ gives the reciprocal, which equals the partition generating function (up to a q-shift).

Parameters

Name	Type	Description
b	Integer	Base: the product runs over q^{bk} for $k \geq 1$. Must be positive.
t	Integer	Exponent: the power to which the eta function is raised. Must be positive.
order	Integer	Truncation order for the result

Examples

```
q> etaq(1, 1, 10)
1 - q - q^2 + q^5 + q^7 + O(q^10)
```

```
q> etaq(2, 1, 10)
1 - q^2 - q^3 - q^4 + q^7 + q^8 + q^9 + O(q^10)
```

```
q> etaq(1, 24, 10)
1 - q + O(q^10)
```

Edge Cases and Constraints

- b must be a positive integer.
- t must be a positive integer. For negative exponents (eta quotients), use `aqprod` directly or compose products.
- The q-shift $q^{bt/24}$ is included automatically as a rational power.
- `etaq(1, 1, order)` reproduces the Euler function $\prod(1-q^k)$, matching `aqprod(1, 1, 1, infinity, order)`.

Related: aqprod, etamake, prove_eta_id, qetamake

5.1.4 jacprod

```
jacprod(a, b, order)
```

Compute the Jacobi triple product $J(a,b)$. This product appears throughout the theory of theta functions, modular forms, and partition identities. It is the product side of the celebrated Jacobi triple product identity.

Mathematical Definition

$$J(a,b) = \prod_{k\geq 1}\left(1-q^{bk}\right)\left(1-q^{bk-a}\right)\left(1-q^{b(k-1)+a}\right)$$

Equivalently, this equals the Jacobi theta function $\sum_{n=-\infty}^{\infty}(-1)^n q^{b\binom{n}{2}+an}$ by the Jacobi triple product identity.

Parameters

Name	Type	Description
a	Integer	Residue parameter (the “shift” in the product)
b	Integer	Period parameter (spacing between factors)
order	Integer	Truncation order for the result

Examples

```
q> jacprod(1, 2, 10)
1 - 2*q + 2*q^4 - 2*q^9 + O(q^10)
```

```
q> jacprod(1, 3, 15)
1 - q - q^2 + q^5 + q^7 - q^12 + O(q^15)
```

```
q> jacprod(1, 5, 15)
1 - q - q^4 + q^7 + q^13 + O(q^15)
```

Edge Cases and Constraints

- b must be a positive integer.
- a must satisfy $0 < a < b$ for a well-defined product.
- `jacprod(1, 2, order)` produces the theta function $\theta_4(q)$.

Related: tripleprod, jacprodmake, theta2, theta3, theta4

5.1.5 tripleprod

```
tripleprod(coeff_num, coeff_den, power, order)
```

Compute the Jacobi triple product in its $(a;q)$-factored form, where $a = (\text{coeff\_num} / \text{coeff\_den}) \cdot q^{\text{power}}$. This is the product of three infinite q-Pochhammer symbols and appears in many partition and theta function identities.

Mathematical Definition

$$\text{tripleprod}(a) = (a;q)_\infty \cdot (q/a;q)_\infty \cdot (q;q)_\infty$$

where $a = (c_1/c_2) \cdot q^p$ and the three factors are evaluated via `aqprod`. Note that if $a = q^k$ for $k \geq 0$, the first factor $(a;q)_\infty$ contains the term $\left(1 - q^k \cdot q^k\right)\cdots$ and the overall product is well-defined but may evaluate to zero if a is a power of q (since $\left(q^k;q\right)_\infty = 0$ when k is a non-negative integer due to the factor $\left(1 - q^k \cdot q^0\right) = 0$ at $k = 0$, etc.).

Parameters

Name	Type	Description
coeff_num	Integer	Numerator of the coefficient a
coeff_den	Integer	Denominator of the coefficient a (must be nonzero)
power	Integer	Power of q in $a = (c_1/c_2)q^{\text{power}}$
order	Integer	Truncation order for the result

Examples

```
q> tripleprod(1, 1, 1, 10)
O(q^10)
```

```
q> jacprod(1, 2, 10)
1 - 2*q + 2*q^4 - 2*q^9 + O(q^10)
```

Edge Cases and Constraints

- `coeff_den` must be nonzero.
- When $a = q^k$ for a non-negative integer k, the factor $(q/a;q)_\infty = \left(q^{1-k};q\right)_\infty$ vanishes (containing the term $1 - q^0 = 0$ when $k \geq 1$, or $1 - 1 = 0$ when $k = 0$). In particular, `tripleprod(1, 1, 1, order)` yields 0. Use `jacprod` for non-trivial triple products.
- The triple product is related to `jacprod` by a change of parametrization.

Related: aqprod, jacprod, quinprod, theta4

5.1.6 quinprod

```
quinprod(coeff_num, coeff_den, power, order)
```

Compute the quintuple product identity expansion. The quintuple product is a product of five infinite q-Pochhammer symbols and appears in advanced partition identities and modular form theory.

Mathematical Definition

$$\text{quinprod}(a) = (a;q)_\infty (q/a;q)_\infty (a^2;q^2)_\infty (q^2/a^2;q^2)_\infty (q;q)_\infty$$

where $a = (c_1/c_2) \cdot q^p$. This contains the triple product $(a;q)_\infty (q/a;q)_\infty (q;q)_\infty$ as a factor, with two additional terms involving a^2 and q^2.

Parameters

Name	Type	Description
coeff_num	Integer	Numerator of the coefficient a
coeff_den	Integer	Denominator of the coefficient a (must be nonzero)
power	Integer	Power of q in $a = (c_1/c_2)q^{\text{power}}$
order	Integer	Truncation order for the result

Examples

```
q> quinprod(1, 1, 1, 10)
1 - q - q^2 + q^5 + q^7 + O(q^10)
```

Edge Cases and Constraints

- `coeff_den` must be nonzero.
- Like `tripleprod`, evaluates to zero when a is a non-negative integer power of q (due to vanishing q-Pochhammer factors).
- Shares the Euler-function factor $(q;q)_\infty$ with `tripleprod`.

Related: tripleprod, aqprod, jacprod

5.1.7 winquist

```
winquist(a_cn, a_cd, a_p, b_cn, b_cd, b_p, order)
```

Compute the Winquist product with two base parameters $a = (a_{\text{cn}}/a_{\text{cd}}) \cdot q^{a_p}$ and $b = (b_{\text{cn}}/b_{\text{cd}}) \cdot q^{b_p}$. The Winquist product is a product of 10 theta-type factors used primarily in partition congruence proofs, particularly for Ramanujan's congruence $p(11n + 6) \equiv 0 \pmod{11}$.

Mathematical Definition

The Winquist product is defined as a product of 10 modified theta functions. With a and b as above, it takes the form:

$$W(a, b) = \prod_{10 \text{ factors}} \text{theta-type}(a, b, q)$$

Each factor is of the form $(x;q)_\infty$ for various combinations of a, b, q/a, q/b, ab, $q/(ab)$, a/b, and qb/a.

Parameters

Name	Type	Description
a_cn	Integer	Numerator of coefficient a
a_cd	Integer	Denominator of coefficient a (must be nonzero)
a_p	Integer	Power of q in $a = (a_{\text{cn}}/a_{\text{cd}})q^{a_p}$
b_cn	Integer	Numerator of coefficient b
b_cd	Integer	Denominator of coefficient b (must be nonzero)
b_p	Integer	Power of q in $b = (b_{\text{cn}}/b_{\text{cd}})q^{b_p}$
order	Integer	Truncation order for the result

Examples

```
q> winquist(1, 1, 1, 1, 1, 2, 10)
O(q^10)
```

Edge Cases and Constraints

- Both `a_cd` and `b_cd` must be nonzero.
- Used primarily in partition congruence proofs; typically called with carefully chosen parameters derived from modular arithmetic.
- Takes 7 parameters: two (c_n, c_d, p) triples for a and b, plus the truncation order.

Related: aqprod, tripleprod, prove_eta_id

6 Partitions

An integer partition of n is a way of writing n as a sum of positive integers, where order does not matter. For example, the 7 partitions of 5 are: 5, $4+1$, $3+2$, $3+1+1$, $2+2+1$, $2+1+1+1$, $1+1+1+1+1$. The number of partitions of n is denoted $p(n)$.

Partition generating functions encode the sequence $\{p(n)\}$ (and its variants) as formal power series in q. These generating functions are intimately connected to infinite products via Euler's identity:

$$\sum_{n\geq 0} p(n)q^n = \prod_{k\geq 1} \frac{1}{1-q^k}$$

q-Kangaroo provides seven partition functions: one that returns a single integer $p(n)$, and six that return generating functions as formal power series in q, including generating functions for distinct parts, odd parts, bounded parts, rank, and crank statistics.

6.1 Function Reference

6.1.1 partition_count

```
partition_count(n)
```

Compute the number of partitions $p(n)$ of the non-negative integer n. Unlike the other partition functions, this returns a single integer rather than a power series. Internally, it uses the pentagonal number recurrence for efficient computation.

Mathematical Definition

$$p(n) = \text{number of partitions of } n$$

Computed via the recurrence derived from Euler's pentagonal number theorem:

$$p(n) = \sum_{k\neq 0} (-1)^{k+1} p(n - k(3k-1)/2)$$

with $p(0) = 1$ and $p(n) = 0$ for $n < 0$.

Parameters

Name	Type	Description
n	Integer	The non-negative integer to partition

Examples

```
q> partition_count(5)
7
```

```
q> partition_count(10)
42
```

```
q> partition_count(100)
190569292
```

```
q> partition_count(200)
3972999029388
```

Edge Cases and Constraints

- n must be a non-negative integer.
- Returns an integer, not a series. To get the generating function, use `partition_gf`.
- Uses exact arbitrary-precision integer arithmetic, so large values of n are supported.

Related: partition_gf, distinct_parts_gf, odd_parts_gf

6.1.2 partition_gf

```
partition_gf(order)
```

Compute the partition generating function $\sum_{n \geq 0} p(n) q^n$ as a formal power series truncated to the given order. The coefficient of q^n in the result is the number of partitions of n.

Mathematical Definition

$$\sum_{n \geq 0} p(n) q^n = \frac{1}{(q;q)_\infty} = \prod_{k \geq 1} \frac{1}{1 - q^k}$$

This is the reciprocal of the Euler function $(q;q)_\infty$.

Parameters

Name	Type	Description
order	Integer	Truncation order for the result

Examples

```
q> partition_gf(10)
1 + q + 2*q^2 + 3*q^3 + 5*q^4 + 7*q^5 + 11*q^6 + 15*q^7 + 22*q^8 + 30*q^9 + O(q^10)
```

Edge Cases and Constraints

- `order` must be a positive integer.
- The series begins $1 + q + 2q^2 + 3q^3 + 5q^4 + \cdots$, matching the well-known partition numbers.
- At `order = 1`, returns just `1 + O(q)`.

Related: partition_count, distinct_parts_gf, odd_parts_gf, bounded_parts_gf, aqprod

6.1.3 distinct_parts_gf

```
distinct_parts_gf(order)
```

Compute the generating function for partitions into distinct parts. The coefficient of q^n counts the number of partitions of n in which all parts are different.

Mathematical Definition

$$(-q;q)_\infty = \prod_{k\geq 1}\left(1+q^k\right)$$

Each factor $\left(1+q^k\right)$ says: include part k at most once.

Parameters

Name	Type	Description
order	Integer	Truncation order for the result

Examples

```
q> distinct_parts_gf(10)
1 + q + q^2 + 2*q^3 + 2*q^4 + 3*q^5 + 4*q^6 + 5*q^7 + 6*q^8 + 8*q^9 + O(q^10)
```

Edge Cases and Constraints

- `order` must be a positive integer.
- By Euler's theorem, this equals `odd_parts_gf(order)` – partitions into distinct parts are equinumerous with partitions into odd parts.

Related: odd_parts_gf, partition_gf, mprodmake

6.1.4 odd_parts_gf

```
odd_parts_gf(order)
```

Compute the generating function for partitions into odd parts. The coefficient of q^n counts the number of partitions of n in which every part is odd (1, 3, 5, 7, …).

Mathematical Definition

$$\frac{1}{(q;q^2)_\infty} = \prod_{k\geq 0} \frac{1}{1-q^{2k+1}}$$

By Euler's partition theorem, this equals the distinct-parts generating function: $(-q;q)_\infty$.

Parameters

Name	Type	Description
order	Integer	Truncation order for the result

Examples

```
q> odd_parts_gf(10)
1 + q + q^2 + 2*q^3 + 2*q^4 + 3*q^5 + 4*q^6 + 5*q^7 + 6*q^8 + 8*q^9 + O(q^10)
```

Edge Cases and Constraints

- `order` must be a positive integer.
- The output is identical to `distinct_parts_gf(order)`, confirming Euler's theorem computationally.

Related: distinct_parts_gf, partition_gf, bounded_parts_gf

6.1.5 bounded_parts_gf

```
bounded_parts_gf(max_part, order)
```

Compute the generating function for partitions whose largest part is at most `max_part`. The coefficient of q^n counts partitions of n with all parts $\leq m$.

Mathematical Definition

$$\prod_{k=1}^{m} \frac{1}{1-q^k}$$

where m is `max_part`. This is a finite product (a rational function of q), unlike the infinite-product partition generating function.

Parameters

Name	Type	Description
max_part	Integer	Maximum allowed part size (must be positive)
order	Integer	Truncation order for the result

Examples

```
q> bounded_parts_gf(3, 10)
1 + q + 2*q^2 + 3*q^3 + 4*q^4 + 5*q^5 + 7*q^6 + 8*q^7 + 10*q^8 + 12*q^9 + O(q^10)
```

```
q> bounded_parts_gf(3, 15)
1 + q + 2*q^2 + 3*q^3 + 4*q^4 + 5*q^5 + 7*q^6 + 8*q^7 + 10*q^8 + 12*q^9 + 14*q^10 + 16*q^11 + 19*q^12 +
21*q^13 + 24*q^14 + O(q^15)
```

Edge Cases and Constraints

- `max_part` must be a positive integer.
- As `max_part` increases, the result approaches `partition_gf(order)`.
- `bounded_parts_gf(1, order)` returns the all-ones series $1 + q + q^2 + \cdots + O(q^{\text{order}})$.

Related: partition_gf, qbin, aqprod

6.1.6 rank_gf

```
rank_gf(z_num, z_den, order)
```

Compute the rank generating function $R(z;q)$ where $z = z_{\text{num}}/z_{\text{den}}$. The rank of a partition is defined as (largest part) − (number of parts). The coefficient of $z^m q^n$ in $R(z;q)$ counts partitions of n with rank m. At $z = 1$, the rank generating function reduces to the ordinary partition generating function.

Mathematical Definition

$$R(z;q) = \sum_{n \geq 0} \sum_{m} N(m,n) z^m q^n$$

where $N(m,n)$ counts partitions of n with rank m. The rank was conjectured by Dyson (1944) to explain Ramanujan's partition congruences modulo 5 and 7.

Parameters

Name	Type	Description
z_num	Integer	Numerator of the variable z
z_den	Integer	Denominator of the variable z (must be nonzero)
order	Integer	Truncation order for the result

Examples

```
q> rank_gf(1, 1, 10)
1 + q + 2*q^2 + 3*q^3 + 5*q^4 + 7*q^5 + 11*q^6 + 15*q^7 + 22*q^8 + 30*q^9 + O(q^10)
```

Edge Cases and Constraints

- `z_den` must be nonzero.
- At $z = 1$ (i.e., `rank_gf(1, 1, order)`), the result equals `partition_gf(order)`.
- The rank generating function is useful for studying Ramanujan-type congruences.

Related: crank_gf, partition_gf, sift, findcong

6.1.7 crank_gf

```
crank_gf(z_num, z_den, order)
```

Compute the crank generating function $C(z; q)$ where $z = z_{\text{num}}/z_{\text{den}}$. The crank of a partition was introduced by Andrews and Garvan (1988) to explain Ramanujan's partition congruence modulo 11. Like the rank generating function, at $z = 1$ it reduces to the partition generating function.

Mathematical Definition

$$C(z; q) = \prod_{n \geq 1} \frac{(1 - q^n)^2}{(1 - zq^n)(1 - q^n/z)}$$

At $z = 1$, the numerator and denominator factors cancel, giving $1/(q; q)_\infty$.

Parameters

Name	Type	Description
z_num	Integer	Numerator of the variable z
z_den	Integer	Denominator of the variable z (must be nonzero)
order	Integer	Truncation order for the result

Examples

```
q> crank_gf(1, 1, 10)
1 + q + 2*q^2 + 3*q^3 + 5*q^4 + 7*q^5 + 11*q^6 + 15*q^7 + 22*q^8 + 30*q^9 + O(q^10)
```

Edge Cases and Constraints

- `z_den` must be nonzero.
- At $z = 1$ (i.e., `crank_gf(1, 1, order)`), the result equals `partition_gf(order)`.
- The crank statistic explains all three of Ramanujan's congruences $p(5n + 4) \equiv 0 \pmod 5$, $p(7n + 5) \equiv 0 \pmod 7$, and $p(11n + 6) \equiv 0 \pmod{11}$.

Related: rank_gf, partition_gf, sift, findcong

7 Theta Functions

The Jacobi theta functions θ_2, θ_3, and θ_4 are fundamental objects in number theory, combinatorics, and mathematical physics. They arise as generating functions for sums of squares, as building blocks for modular forms, and in the theory of elliptic functions.

q-Kangaroo implements the three "one-variable" Jacobi theta functions in the q-notation convention. Each is expressible both as a sum over integers and as an infinite product via the Jacobi triple product identity.

A classical identity connecting all three theta functions is the **Jacobi identity**:

$$\theta_3(q)^4 = \theta_2(q)^4 + \theta_4(q)^4$$

This identity can be verified computationally in q-Kangaroo by comparing the power series expansions of both sides.

7.1 Function Reference

7.1.1 theta2

```
theta2(order)
```

Compute the Jacobi theta function $\theta_2(q)$. This function uses the standard q-convention where the series is expressed in integer powers of q (absorbing the classical $q^{1/4}$ prefactor into the series variable).

Mathematical Definition

Sum form:

$$\theta_2(q) = 2q^{1/4} \sum_{n \geq 0} q^{n(n+1)}$$

In q-Kangaroo's integer-power convention, the series is:

$$\theta_2(q) = 2q + 2q^9 + 2q^{25} + \cdots$$

(absorbing the $q^{1/4}$ factor).

Product form:

$$\theta_2(q) = 2q^{1/4} \prod_{k \geq 1} \left(1 - q^{2k}\right)\left(1 + q^{2k}\right)^2$$

Parameters

Name	Type	Description
order	Integer	Truncation order for the result

Examples

```
q> theta2(10)
2*q + 2*q^9 + O(q^10)
```

```
q> theta2(50)
2*q + 2*q^9 + 2*q^25 + 2*q^49 + O(q^50)
```

Edge Cases and Constraints

- `order` must be a positive integer.
- The nonzero terms appear at q^{n^2} for odd n (i.e., $q^1, q^9, q^{25}, q^{49}, \cdots$) in the integer-power convention.
- For small orders, the series may have very few nonzero terms (e.g., only $2q$ for `order` ≤ 9).

Related: theta3, theta4, jacprod

7.1.2 theta3

```
theta3(order)
```

Compute the Jacobi theta function $\theta_3(q)$. This is the generating function for representations as sums of squares: the coefficient of q^n in $\theta_3(q)^k$ counts the number of ways to write n as a sum of k squares (with sign and order).

Mathematical Definition

Sum form:

$$\theta_3(q) = 1 + 2\sum_{n\geq 1} q^{n^2} = \sum_{n=-\infty}^{\infty} q^{n^2}$$

Product form (Jacobi triple product):

$$\theta_3(q) = \prod_{k\geq 1}(1-q^{2k})\left(1+q^{2k-1}\right)^2$$

Parameters

Name	Type	Description
order	Integer	Truncation order for the result

Examples

```
q> theta3(10)
1 + 2*q + 2*q^4 + 2*q^9 + O(q^10)
```

```
q> qdegree(theta3(10))
9
```

```
q> lqdegree(theta3(10))
0
```

Edge Cases and Constraints

- `order` must be a positive integer.
- The nonzero terms appear at perfect squares: $q^0, q^1, q^4, q^9, q^{16}, q^{25}, \cdots$
- $\theta_3(q)^2$ is the generating function for sums of two squares (Jacobi's two-square theorem).

Related: theta2, theta4, jacprod

7.1.3 theta4

```
theta4(order)
```

Compute the Jacobi theta function $\theta_4(q)$. This is related to θ_3 by $\theta_4(q) = \theta_3(-q)$, i.e., the series obtained by the substitution $q \to -q$.

Mathematical Definition

Sum form:

$$\theta_4(q) = 1 + 2\sum_{n\geq 1}(-1)^n q^{n^2}$$

Product form:

$$\theta_4(q) = \prod_{k\geq 1}\left(1-q^{2k}\right)\left(1-q^{2k-1}\right)^2$$

Equivalently, $\theta_4(q) = J(1,2) =$ `jacprod(1, 2, order)`.

Parameters

Name	Type	Description
order	Integer	Truncation order for the result

Examples

```
q> theta4(10)
1 - 2*q + 2*q^4 - 2*q^9 + O(q^10)
```

```
q> jacprod(1, 2, 10)
1 - 2*q + 2*q^4 - 2*q^9 + O(q^10)
```

Edge Cases and Constraints

- `order` must be a positive integer.
- Related to `theta3` by $\theta_4(q) = \theta_3(-q)$: coefficients at odd-index squares are negated.
- `jacprod(1, 2, order)` produces the same series as `theta4(order)`.

Related: theta3, theta2, jacprod

7.2 The Jacobi Identity

The Jacobi identity $\theta_3^4 = \theta_2^4 + \theta_4^4$ can be verified computationally by comparing the two sides as truncated power series:

```
q> a := theta3(50)^4:
q> b := theta2(50)^4 + theta4(50)^4:
q> a - b
O(q^50)
```

The vanishing of the difference to high order provides strong numerical evidence for the identity; for a rigorous proof, use `prove_eta_id` on the corresponding eta-quotient formulation.

8 Series Analysis

The series analysis functions provide tools for dissecting, factoring, and reverse-engineering power series. Given an unknown or computed series, these functions help researchers discover its structure: Is it an eta quotient? A Jacobi product? A $(1+q^n)$ product? What are its extremal degrees?

These are the "detective tools" of q-series research. While the product and partition functions **construct** series from known definitions, the series analysis functions work in the opposite direction – they take a series and **decompose** it into recognizable forms.

8.1 Function Reference

8.1.1 sift

```
sift(series, m, j)
```

Extract the arithmetic subsequence of coefficients from a power series. Given a series $f(q) = \sum a_n q^n$, the call `sift(f, m, j)` returns a new series whose n-th coefficient is the $(mn+j)$-th coefficient of the input. This is the fundamental tool for studying partition congruences and arithmetic properties of q-series coefficients.

Mathematical Definition

Given $f(q) = \sum_{n \geq 0} a_n q^n$, the sifted series is:

$$\text{sift}(f, m, j) = \sum_{n \geq 0} a_{mn+j} q^n$$

This extracts every m-th coefficient, starting from position j.

Parameters

Name	Type	Description
series	Series	The input power series to sift
m	Integer	The modulus (step size); must be a positive integer
j	Integer	The residue (starting offset); must satisfy $0 \leq j < m$

Examples

```
q> sift(partition_gf(50), 5, 4)
5 + 30*q + 135*q^2 + 490*q^3 + 1575*q^4 + 4565*q^5 + 12310*q^6 + 31185*q^7 + 75175*q^8 + 173525*q^9
+ O(q^10)
```

Edge Cases and Constraints

- m must be a positive integer.
- j must satisfy $0 \le j < m$.
- The output series has approximately $\lfloor (\text{order} - j)/m \rfloor$ terms from the original.
- The example demonstrates Ramanujan's congruence: all coefficients of `sift(partition_gf(N), 5, 4)` are divisible by 5, since $p(5n+4) \equiv 0 \pmod 5$.

Related: findcong, partition_gf, rank_gf

8.1.2 qdegree

```
qdegree(series)
```

Return the highest power of q with a nonzero coefficient in the series. For truncated series, this is bounded by one less than the truncation order. This is a utility function for quickly inspecting the support of a series.

Parameters

Name	Type	Description
series	Series	The input power series

Examples

```
q> qdegree(theta3(10))
9
```

```
q> qdegree(partition_gf(10))
9
```

Edge Cases and Constraints

- For the zero series, the behavior depends on the truncation order.
- For polynomial series (no truncation), returns the true degree.
- Paired with `lqdegree` to determine the full support range.

Related: lqdegree, qfactor

8.1.3 lqdegree

```
lqdegree(series)
```

Return the lowest power of q with a nonzero coefficient in the series. For series beginning with the constant term 1, this returns 0. For series like $q + q^2 + \cdots$, this returns 1.

Parameters

Name	Type	Description
series	Series	The input power series

Examples

```
q> lqdegree(theta3(10))
0
```

```
q> lqdegree(theta2(10))
1
```

Edge Cases and Constraints

- For the zero series, returns 0.
- Also known as the q-adic valuation of the series.
- Paired with `qdegree` to determine the full support range.

Related: qdegree, qfactor

8.1.4 qfactor

```
qfactor(series)
```

Factor a polynomial series into $(1 - q^i)$ factors by top-down division. Returns a dictionary mapping each factor $(1 - q^i)$ to its multiplicity, along with a scalar coefficient and an `is_exact` flag indicating whether the factorization is complete.

Parameters

Name	Type	Description
series	Series	The input series to factor (should be a polynomial for exact results)

Examples

```
q> qfactor(aqprod(1, 1, 1, 5, 20))
{scalar: 1, factors: {1: 1, 2: 1, 3: 1, 4: 1, 5: 1}, is_exact: true}
```

Edge Cases and Constraints

- Works best on polynomial series (finite q-Pochhammer products).
- For truncated infinite series, the factorization may be approximate (`is_exact: false`).
- The scalar field captures any leading constant factor.
- The result `{1: 1, 2: 1, 3: 1, 4: 1, 5: 1}` means $(1-q)(1-q^2)(1-q^3)(1-q^4)(1-q^5)$.

Related: prodmake, etamake, aqprod

8.1.5 prodmake

```
prodmake(series, max_n)
```

Find the infinite product representation of a series via the logarithmic derivative method. Returns exponents a_n such that the series equals $\prod_{n\geq 1}(1-q^n)^{a_n}$ (up to the truncation order of the input series).

Mathematical Definition

Given $f(q)$, find integers a_n such that:

$$f(q) = \prod_{n\geq 1}(1-q^n)^{a_n}$$

The algorithm takes the logarithmic derivative f'/f, expands in a power series, and reads off the exponents from the resulting Dirichlet-type series.

Parameters

Name	Type	Description
series	Series	The input power series to decompose
max_n	Integer	Maximum index n to compute exponents for

Examples

```
q> prodmake(partition_gf(50), 20)
{exponents: {1: 1, 2: 1, 3: 1, ..., 20: 1}, terms_used: 20}
```

Edge Cases and Constraints

- `max_n` should be significantly less than the truncation order of the input series.
- All exponents for `partition_gf` are 1 (since $1/(q;q)_\infty = \prod 1/(1-q^n)$), confirming the product form.
- The method is numerical: it matches coefficients, not a symbolic proof. Use `prove_eta_id` for rigorous verification.

Related: etamake, jacprodmake, mprodmake, qetamake, qfactor

8.1.6 etamake

```
etamake(series, max_n)
```

Find an eta-quotient representation of the series via Mobius inversion. An eta quotient is a product of Dedekind eta functions $\prod_d \eta(d\tau)^{r_d}$. The function returns the divisor-grouped exponents r_d and any q-shift.

Mathematical Definition

Given $f(q)$, find integers r_d and a rational q-shift s such that:

$$f(q) = q^s \prod_d \eta(d\tau)^{r_d}$$

where $\eta(d\tau) = q^{d/24} \prod_{k\geq 1}(1-q^{dk})$. The exponents are found by applying Mobius inversion to the output of `prodmake`.

Parameters

Name	Type	Description
series	Series	The input power series to decompose
max_n	Integer	Maximum divisor to search for eta factors

Examples

```
q> etamake(partition_gf(50), 10)
{factors: {1: -1}, q_shift: -1/24}
```

Edge Cases and Constraints

- The result `{1: -1}` means $\eta(\tau)^{-1} = 1/\eta(\tau)$, confirming that `partition_gf` $= q^{-1/24}/\eta(\tau)$.
- `max_n` should be less than half the truncation order for reliable results.
- Not every series has an eta-quotient form; the result is a best-effort approximation.

Related: etaq, prodmake, qetamake, prove_eta_id

8.1.7 jacprodmake

```
jacprodmake(series, max_n)
```

Find a Jacobi product representation of the series with automatic period search and residue grouping. Returns the period, grouped residue exponents, a scalar, and an `is_exact` flag indicating whether the product matches the input series exactly.

Mathematical Definition

Given $f(q)$, find a period b and residue exponents such that:

$$f(q) \approx \prod_{(a,b)\in \text{ residues}} J(a,b)^{e_{a,b}}$$

where $J(a, b)$ is the Jacobi triple product. The algorithm searches over candidate periods and uses the `prodmake` exponents grouped by residue classes modulo b.

Parameters

Name	Type	Description
series	Series	The input power series to decompose
max_n	Integer	Maximum period to search

Examples

```
q> jacprodmake(theta3(50), 10)
{factors: {(1,8): -2, (2,8): 3, (3,8): -2}, scalar: 1, is_exact: false}
```

Edge Cases and Constraints

- `is_exact: true` means the Jacobi product reproduces the input series exactly (within truncation).
- `is_exact: false` means the best approximation was found but does not match exactly.
- More input terms (higher truncation order) improve the reliability of the decomposition.

Related: jacprod, prodmake, etamake

8.1.8 mprodmake

```
mprodmake(series, max_n)
```

Find a $(1 + q^n)$ product representation by iterative extraction. Returns exponents b_n such that the series equals $\prod_{n \geq 1} (1 + q^n)^{b_n}$. Internally converts between $(1 + q^n)$ and $(1 - q^{2n})/(1 - q^n)$ representations.

Mathematical Definition

Given $f(q)$, find integers b_n such that:

$$f(q) = \prod_{n \geq 1} (1 + q^n)^{b_n}$$

Since $(1 + q^n) = (1 - q^{2n})/(1 - q^n)$, this is a rearrangement of the $(1 - q^n)$ product form.

Parameters

Name	Type	Description
series	Series	The input power series to decompose
max_n	Integer	Maximum index n to compute exponents for

Examples

```
q> mprodmake(distinct_parts_gf(50), 10)
{1: 1, 2: 1, 3: 1, 4: 1, 5: 1, 6: 1, 7: 1, 8: 1, 9: 1, 10: 1}
```

Edge Cases and Constraints

- All exponents for `distinct_parts_gf` are 1, confirming $(-q;q)_\infty = \prod(1+q^k)$.
- `max_n` should be significantly less than the truncation order.
- Series that are not $(1+q^n)$ products will still produce exponents, but they represent only an approximation.

Related: distinct_parts_gf, prodmake, etamake

8.1.9 qetamake

```
qetamake(series, max_n)
```

Find a combined eta/q-Pochhammer product representation. This extends `etamake` by also searching for additional q-Pochhammer factors beyond pure eta quotients. Returns eta factors and any q-shift.

Parameters

Name	Type	Description
series	Series	The input power series to decompose
max_n	Integer	Maximum index to search

Examples

```
q> qetamake(partition_gf(50), 10)
{factors: {1: -1}, q_shift: 0}
```

Edge Cases and Constraints

- For pure eta quotients, the output matches `etamake` (possibly with a different q-shift convention).
- For series with non-eta factors, `qetamake` may find a representation where `etamake` cannot.
- `max_n` should be less than half the truncation order for reliable results.

Related: etamake, prodmake, etaq, aqprod

9 Relation Discovery

These functions use Gaussian elimination over $\mathbb{Q}$ (or $\mathbb{Z}/p\mathbb{Z}$) to discover algebraic relations among q-series. They are the primary research tools for finding new identities. All functions operate on truncated series and compare coefficients up to the truncation order. Several functions accept a `topshift` parameter that controls how many leading coefficients to skip – useful when series have known leading terms that should not participate in the relation search.

9.1 Linear Combinations

These functions find explicit coefficient vectors expressing a target series as a polynomial combination of candidate series.

9.1.1 findlincombo

```
findlincombo(target, [candidates], topshift)
```

Find rational coefficients c_i such that

$$\text{target} = \sum_i c_i \cdot \text{candidates}[i].$$

Uses exact arithmetic over $\mathbb{Q}$ via reduced row echelon form (RREF). This is the main workhorse for discovering linear relations among q-series: given a target series and a list of candidate series, it determines whether the target can be written as a rational linear combination of the candidates, and if so returns the coefficients.

Parameters

Name	Type	Description
target	Series	The series to express as a linear combination
candidates	List of Series	List of candidate series to combine
topshift	Integer	Number of leading coefficients to skip

Examples

```
q> findlincombo(partition_gf(30), [distinct_parts_gf(30), odd_parts_gf(30)], 0)
[0, 1]
```

```
q> findlincombo(theta3(50)^2, [theta2(50)^2, theta4(50)^2], 0)
null
```

Edge Cases and Constraints

- Returns `null` if no linear combination exists within the truncation order.
- The candidates list must be non-empty.
- Coefficients are exact rationals, not floating-point approximations.

Related: findhomcombo, findnonhomcombo, findlincombomodp

9.1.2 findhomcombo

```
findhomcombo(target, [candidates], degree, topshift)
```

Find a homogeneous polynomial combination of the given degree matching the target. Generates all monomials of exact degree d in the candidate series, then applies RREF over $\mathbb{Q}$ to find coefficients. For example, with degree 2 and candidates $[f, g, h]$, the search space includes f^2, fg, fh, g^2, gh, h^2.

Parameters

Name	Type	Description
target	Series	The target series to match
candidates	List of Series	Candidate series for building monomials
degree	Integer	Degree of the homogeneous polynomial
topshift	Integer	Number of leading coefficients to skip

Examples

```
q> findhomcombo(theta3(50)^4, [theta2(50)^2, theta4(50)^2], 2, 0)
homogeneous degree-2 coefficients
```

Edge Cases and Constraints

- Returns `null` if no homogeneous combination of the given degree exists.
- The number of monomials grows with $\binom{n+d-1}{d}$ where n is the number of candidates.

Related: findlincombo, findnonhomcombo, findhomcombomodp

9.1.3 findnonhomcombo

```
findnonhomcombo(target, [candidates], degree, topshift)
```

Find a nonhomogeneous polynomial combination up to the given degree matching the target. Includes all monomials from degree 0 (constant term) through degree d in the candidate series, providing a more flexible search than `findhomcombo`.

Parameters

Name	Type	Description
target	Series	The target series to match
candidates	List of Series	Candidate series for building monomials
degree	Integer	Maximum degree of the polynomial
topshift	Integer	Number of leading coefficients to skip

Examples

```
q> findnonhomcombo(partition_gf(30), [etaq(1,1,30), etaq(2,1,30)], 3, 0)
polynomial combination coefficients
```

Edge Cases and Constraints

- Returns `null` if no polynomial combination up to the given degree exists.
- Includes the constant monomial (degree 0), so this subsumes linear search.

Related: findlincombo, findhomcombo

9.1.4 findlincombomodp

```
findlincombomodp(target, [candidates], p, topshift)
```

Find a linear combination matching the target with arithmetic performed modulo a prime p. Uses Fermat's little theorem for modular inverse ($a^{-1} \equiv a^{p-2} \bmod p$). This is useful when exact rational arithmetic is too expensive – for instance, when working with very long series where rational coefficient growth causes slowdown.

Parameters

Name	Type	Description
target	Series	The target series
candidates	List of Series	Candidate series to combine
p	Integer	A prime modulus
topshift	Integer	Number of leading coefficients to skip

Examples

```
q> findlincombomodp(partition_gf(100), [odd_parts_gf(100)], 7, 0)
coefficients in Z/7Z
```

Edge Cases and Constraints

- The modulus p must be prime for Fermat inverse to work correctly.
- Results are only valid modulo p – a match mod p does not guarantee an exact rational relation.

Related: findlincombo, findhomcombomodp

9.1.5 findhomcombomodp

```
findhomcombomodp(target, [candidates], p, degree, topshift)
```

Find a homogeneous polynomial combination of the given degree, with arithmetic performed modulo a prime p. Combines the monomial generation of `findhomcombo` with the modular arithmetic of `findlincombomodp`.

Parameters

Name	Type	Description
target	Series	The target series
candidates	List of Series	Candidate series for building monomials
p	Integer	A prime modulus
degree	Integer	Degree of the homogeneous polynomial
topshift	Integer	Number of leading coefficients to skip

Examples

```
q> findhomcombomodp(theta3(50)^4, [theta2(50)^2, theta4(50)^2], 5, 2, 0)
polynomial combination coefficients mod 5
```

Edge Cases and Constraints

- The modulus p must be prime.
- Useful for large-scale searches where exact arithmetic is prohibitive.

Related: findhomcombo, findlincombomodp

9.2 Relation Finding

These functions search for polynomial relations *among* a list of series (without a distinguished target). They return the null space of the coefficient matrix – that is, any nonzero vector in the kernel represents a relation.

9.2.1 findhom

```
findhom([series], degree, topshift)
```

Find a homogeneous polynomial relation of the given degree among the series in the input list. Constructs all monomials of degree d from the input series, builds a coefficient matrix, and computes its null space via RREF. A nonzero null-space vector corresponds to a polynomial identity among the series.

Parameters

Name	Type	Description
series	List of Series	Series among which to find a relation
degree	Integer	Degree of the homogeneous polynomial relation
topshift	Integer	Number of leading coefficients to skip

Examples

```
q> findhom([theta3(50)^2, theta2(50)^2, theta4(50)^2], 1, 0)
[1, -1, -1]
```

Edge Cases and Constraints

- Returns the null space, which may contain multiple independent relations.
- An empty null space means no relation of the given degree exists.

Related: findnonhom, findhommodp, findhomcombo

9.2.2 findnonhom

```
findnonhom([series], degree, topshift)
```

Find a nonhomogeneous relation of the given degree among the series. Includes constant and lower-degree terms in the monomial basis, so this can discover relations like $f^2 + g - 3 = 0$ that `findhom` would miss.

Parameters

Name	Type	Description
series	List of Series	Series among which to find a relation
degree	Integer	Maximum degree of the polynomial relation
topshift	Integer	Number of leading coefficients to skip

Examples

```
q> findnonhom([theta3(50)^2, theta2(50)^2, theta4(50)^2], 1, 0)
relation coefficients
```

Edge Cases and Constraints

- The constant term (degree 0) is always included in the monomial basis.
- Returns the null space of the augmented coefficient matrix.

Related: findhom, findnonhomcombo

9.2.3 findhommodp

```
findhommodp([series], p, degree, topshift)
```

Find a homogeneous relation among the series with arithmetic performed modulo a prime p. Useful when exact rational arithmetic is too expensive for the size of the series or the degree of the relation.

Parameters

Name	Type	Description
series	List of Series	Series among which to find a relation
p	Integer	A prime modulus
degree	Integer	Degree of the homogeneous polynomial relation
topshift	Integer	Number of leading coefficients to skip

Examples

```
q> findhommodp([theta3(50)^2, theta2(50)^2, theta4(50)^2], 7, 1, 0)
relation coefficients mod 7
```

Edge Cases and Constraints

- The modulus p must be prime.
- A relation mod p suggests but does not prove an exact relation over $\mathbb{Q}$.

Related: findhom, findlincombomodp

9.2.4 findmaxind

```
findmaxind([series], topshift)
```

Find a maximally independent subset of the given series via Gaussian elimination. Returns the indices of the pivot columns in the coefficient matrix, identifying which series form a basis for the span of the full list. This is useful for determining the dimension of a space of modular forms or checking whether a new series is linearly independent of known ones.

Parameters

Name	Type	Description
series	List of Series	Series to test for independence
topshift	Integer	Number of leading coefficients to skip

Examples

```
q> findmaxind([partition_gf(30), odd_parts_gf(30), distinct_parts_gf(30)], 0)
[0, 2]
```

Edge Cases and Constraints

- Returns indices (0-based) of the pivot columns after RREF.
- If all series are independent, returns all indices.
- The returned set is not unique – different orderings may produce different pivot selections.

Related: findhom, findlincombo

9.3 Specialized Searches

These functions perform targeted searches for specific types of identities: product relations, partition congruences, and polynomial relations between two series.

9.3.1 findprod

```
findprod([series], max_coeff, max_exp)
```

Find a product identity among the given series by brute-force search over exponent combinations. Tests

$$\prod_i \text{series}[i]^{e_i}$$

for all integer exponent vectors with $|e_i| \leq \text{max\_coeff}$, checking whether any combination produces $1 + O(q^N)$.

Parameters

Name	Type	Description
series	List of Series	Series to combine as a product
max_coeff	Integer	Maximum absolute value of exponents to test
max_exp	Integer	Maximum exponent range

Examples

```
q> findprod([etaq(1,1,30), etaq(2,1,30)], 3, 2)
exponent vector (if product identity exists)
```

Edge Cases and Constraints

- Returns `null` if no product identity is found within the search bounds.
- Search time grows exponentially with the number of series and `max_coeff`.
- Only detects identities where the product equals $1 + O(q^N)$.

Related: findhom, findpoly

9.3.2 findcong

```
findcong(series, [moduli])
```

Find partition-type congruences by checking whether sifted arithmetic subsequences of the input series vanish modulo specified moduli. For each modulus m and residue r, extracts the subsequence of coefficients at positions $mn + r$ and checks divisibility. Reports all (m, r, p) triples where the sifted coefficients are divisible by the prime p.

This is the tool for rediscovering Ramanujan's celebrated congruences: $p(5n + 4) \equiv 0 \bmod 5$, $p(7n + 5) \equiv 0 \bmod 7$, and $p(11n + 6) \equiv 0 \bmod 11$.

Parameters

Name	Type	Description
series	Series	The generating function to analyze
moduli	List of Integer	List of moduli to test

Examples

```
q> findcong(partition_gf(200), [5, 7, 11])
list of (modulus, residue, prime) triples
```

Edge Cases and Constraints

- The series must have enough terms for the sifted subsequences to be meaningful.
- Only checks divisibility by the moduli in the list, not all primes.

Related: sift, findlincombo

9.3.3 findpoly

```
findpoly(x, y, deg_x, deg_y, topshift)
```

Find a polynomial relation $P(x, y) = 0$ between two series x and y. Searches for a polynomial P of degree at most `deg_x` in x and `deg_y` in y whose evaluation at the two series vanishes to the truncation order.

Parameters

Name	Type	Description
x	Series	First series
y	Series	Second series
deg_x	Integer	Maximum degree in x
deg_y	Integer	Maximum degree in y
topshift	Integer	Number of leading coefficients to skip

Examples

```
q> findpoly(theta3(50)^4, theta2(50)^4 + theta4(50)^4, 1, 1, 0)
polynomial coefficients (if relation exists)
```

Edge Cases and Constraints

- Returns `null` if no polynomial relation of the given degree bounds exists.
- Higher degree bounds increase both the chance of finding a relation and the computation time.

Related: findhom, findnonhom, findprod

10 Basic Hypergeometric Series

The basic hypergeometric series ${}_r\varphi_s$ and the bilateral basic hypergeometric series ${}_r\psi_s$ are fundamental objects in q-series theory. The general ${}_r\varphi_s$ is defined by

$$ {}_r\varphi_s\left(\begin{matrix} a_1 & a_2 & \dots & a_r \\ b_1 & b_2 & \dots & b_s \end{matrix}; q, z\right) = \sum_{n=0}^{\infty} \frac{(a_1;q)_n (a_2;q)_n \cdots (a_r;q)_n}{(b_1;q)_n (b_2;q)_n \cdots (b_s;q)_n (q;q)_n} \left[(-1)^n q^{\binom{n}{2}}\right]^{1+s-r} z^n $$

where $(a;q)_n = \prod_{k=0}^{n-1}(1-aq^k)$ is the q-Pochhammer symbol. When $r \le s+1$ and $|z|<1$ the series converges absolutely. The factor $\left[(-1)^n q^{\binom{n}{2}}\right]^{1+s-r}$ reduces to 1 for the most common case $r = s+1$ ("balanced" series).

In q-Kangaroo, the upper and lower parameters are each specified as lists of `(num, den, pow)` triples, where each triple represents the parameter $(\text{num}/\text{den}) \cdot q^{\text{pow}}$. The argument z is similarly encoded as `z_num`, `z_den`, `z_pow` representing $z = (z_{\text{num}}/z_{\text{den}}) \cdot q^{z_{\text{pow}}}$.

10.0.1 phi

```
phi(upper_list, lower_list, z_num, z_den, z_pow, order)
```

Evaluate the basic hypergeometric series ${}_r\varphi_s(\text{upper}; \text{lower}; q, z)$ where r and s are determined by the lengths of the parameter lists. Each parameter is a `(num, den, pow)` triple encoding $(\text{num}/\text{den}) \cdot q^{\text{pow}}$.

Mathematical Definition

$$ {}_r\varphi_s\left(\begin{matrix} a_1 & \dots & a_r \\ b_1 & \dots & b_s \end{matrix}; q, z\right) = \sum_{n=0}^{\infty} \frac{\prod_{j=1}^{r} (a_j;q)_n}{\prod_{j=1}^{s} (b_j;q)_n \cdot (q;q)_n} \left[(-1)^n q^{\binom{n}{2}}\right]^{1+s-r} z^n $$

Parameters

Name	Type	Description
upper_list	List of (Int, Int, Int)	Upper parameters as `(num, den, pow)` triples
lower_list	List of (Int, Int, Int)	Lower parameters as `(num, den, pow)` triples
z_num	Integer	Numerator of z
z_den	Integer	Denominator of z
z_pow	Integer	Power of q in z
order	Integer	Truncation order

Examples

```
q> phi([(1,1,1)], [(1,1,2)], 1, 1, 1, 10)
1 + ... + O(q^10)
```

```
q> phi([], [], 1, 1, 0, 10)
1/(q;q)_inf truncated to O(q^10)
```

Edge Cases and Constraints

- Either `upper_list` or `lower_list` may be empty (yielding a ${}_0\varphi_s$ or ${}_r\varphi_0$).
- Division by zero occurs if a lower parameter equals q^k for some $0 \le k < n$, causing $\left(b_j;q\right)_n = 0$.
- The series terminates if an upper parameter equals q^{-m} for a non-negative integer m.

Related: psi, try_summation, heine1, heine2, heine3

10.0.2 psi

```
psi(upper_list, lower_list, z_num, z_den, z_pow, order)
```

Evaluate the bilateral basic hypergeometric series ${}_r\psi_s(\text{upper};\text{lower};q,z)$, which sums over all integers $n \in \mathbb{Z}$ (both positive and negative indices). The bilateral series is defined by

$$ {}^{r}\psi_{s}\left(\begin{matrix} a_1 & \dots & a_r \\ b_1 & \dots & b_s \\ q & z & \end{matrix}\right) = \sum_{n=-\infty}^{\infty} \frac{\prod_{j=1}^{r} \left(a_j;q\right)_n}{\prod_{j=1}^{s} \left(b_j;q\right)_n} z^n $$

where for negative n, the q-Pochhammer symbols are defined via $(a;q)_{-n} = 1/(aq^{-n};q)_n$.

Mathematical Definition

$$ {}^{r}\psi_{s}\left(\begin{matrix} a_1 & \dots & a_r \\ b_1 & \dots & b_s \\ q & z & \end{matrix}\right) = \sum_{n=-\infty}^{\infty} \frac{\prod_{j=1}^{r} \left(a_j;q\right)_n}{\prod_{j=1}^{s} \left(b_j;q\right)_n} z^n $$

Parameters

Name	Type	Description
upper_list	List of (Int, Int, Int)	Upper parameters as `(num, den, pow)` triples
lower_list	List of (Int, Int, Int)	Lower parameters as `(num, den, pow)` triples
z_num	Integer	Numerator of z
z_den	Integer	Denominator of z
z_pow	Integer	Power of q in z
order	Integer	Truncation order

Examples

```
q> psi([(1,1,1)], [(1,1,2)], 1, 1, 1, 10)
bilateral sum truncated to order 10
```

Edge Cases and Constraints

- The bilateral sum includes negative powers of q if z has non-positive q-power.
- Convergence requires $|b_1 \cdots b_s/(a_1 \cdots a_r)| < |z| < 1$ when $r = s$.

Related: phi, try_summation

10.0.3 try_summation

```
try_summation(upper_list, lower_list, z_num, z_den, z_pow, order)
```

Attempt to find a closed-form summation for a basic hypergeometric series by matching against the five classical summation formulas:

1. **q-Gauss sum** – evaluates ${}_2\varphi_1(a, b; c; q, c/(ab))$
2. **q-Vandermonde (q-Chu–Vandermonde)** – evaluates a terminating ${}_2\varphi_1$
3. **q-Saalschutz (q-Pfaff–Saalschutz)** – evaluates a balanced terminating ${}_3\varphi_2$
4. **q-Kummer** – evaluates certain ${}_2\varphi_1$ at $z = -1$
5. **q-Dixon** – evaluates a very-well-poised ${}_4\varphi_3$

Returns a closed-form product expression if a formula applies, or `null` if no known summation matches.

Parameters

Name	Type	Description
upper_list	List of (Int, Int, Int)	Upper parameters as `(num, den, pow)` triples
lower_list	List of (Int, Int, Int)	Lower parameters as `(num, den, pow)` triples
z_num	Integer	Numerator of z
z_den	Integer	Denominator of z
z_pow	Integer	Power of q in z
order	Integer	Truncation order for verification

Examples

```
q> try_summation([(1,1,1), (1,1,2)], [(1,1,3)], 1, 1, 1, 10)
closed-form product (or null if no formula applies)
```

Edge Cases and Constraints

- Returns `null` if no classical summation formula matches – this does not mean no closed form exists.
- The order parameter is used to verify the candidate closed form against the series.

Related: phi, q_gosper

10.1 Heine Transformations

Heine's three classical transformations relate different ${}_2\varphi_1$ series. Given a ${}_2\varphi_1(a,b;c;q,z)$, each transformation rewrites it as a product of infinite q-Pochhammer quotients times a new ${}_2\varphi_1$ with different parameters. These transformations are fundamental tools for simplifying and evaluating basic hypergeometric series.

All three functions share the same signature pattern: they accept the same six parameters as `phi` and return the transformed series. The input must be a valid ${}_2\varphi_1$ (exactly 2 upper parameters and 1 lower parameter).

10.1.1 heine1

```
heine1(upper_list, lower_list, z_num, z_den, z_pow, order)
```

Apply Heine's first transformation to a ${}_2\varphi_1$ series. Transforms ${}_2\varphi_1(a,b;c;q,z)$ into a q-Pochhammer product prefactor times another ${}_2\varphi_1$ with rearranged parameters:

$$ {}^{2}\varphi_1(a,b;c;q,z) = \frac{(b;q)_\infty (az;q)_\infty}{(c;q)_\infty (z;q)_\infty} \cdot {}^{2}\varphi_1(c/b,z;az;q,b) $$

Parameters

Name	Type	Description
upper_list	List of (Int, Int, Int)	Two upper parameters `(a, b)` as triples
lower_list	List of (Int, Int, Int)	One lower parameter `(c)` as a triple
z_num	Integer	Numerator of z
z_den	Integer	Denominator of z
z_pow	Integer	Power of q in z
order	Integer	Truncation order

Examples

```
q> heine1([(1,1,1), (1,1,2)], [(1,1,3)], 1, 1, 1, 10)
(prefactor, transformed_series)
```

Edge Cases and Constraints

- Requires exactly 2 upper and 1 lower parameter (a ${}_2\varphi_1$).

- The prefactor involves infinite products that must converge within the truncation order.

Related: heine2, heine3, phi

10.1.2 heine2

```
heine2(upper_list, lower_list, z_num, z_den, z_pow, order)
```

Apply Heine's second transformation to a ${}_2\varphi_1$ series. A different rearrangement of the ${}_2\varphi_1$ parameters:

$$ {}^2\varphi_1(a,b;c;q,z) = \frac{(abz/c;q)_\infty}{(z;q)_\infty} \cdot {}^2\varphi_1(c/a,c/b;c;q,abz/c) $$

Parameters

Name	Type	Description
upper_list	List of (Int, Int, Int)	Two upper parameters as triples
lower_list	List of (Int, Int, Int)	One lower parameter as a triple
z_num	Integer	Numerator of z
z_den	Integer	Denominator of z
z_pow	Integer	Power of q in z
order	Integer	Truncation order

Examples

```
q> heine2([(1,1,1), (1,1,2)], [(1,1,3)], 1, 1, 1, 10)
(prefactor, transformed_series)
```

Edge Cases and Constraints

- Requires exactly 2 upper and 1 lower parameter.
- The transformed z-argument is abz/c, which may change convergence behavior.

Related: heine1, heine3, phi

10.1.3 heine3

```
heine3(upper_list, lower_list, z_num, z_den, z_pow, order)
```

Apply Heine's third transformation to a ${}_2\varphi_1$ series. Transforms into a ratio of infinite products times a ${}_2\varphi_1$ with yet another parameter arrangement:

$$ {}^2\varphi_1(a,b;c;q,z) = \frac{(az;q)_\infty (bz;q)_\infty}{(c;q)_\infty (z;q)_\infty} \cdot \frac{(c;q)_\infty}{(abz/c;q)_\infty} \cdot {}^2\varphi_1(abz/c,z;bz;q,c/(az)) $$

Parameters

Name	Type	Description
upper_list	List of (Int, Int, Int)	Two upper parameters as triples
lower_list	List of (Int, Int, Int)	One lower parameter as a triple
z_num	Integer	Numerator of z
z_den	Integer	Denominator of z
z_pow	Integer	Power of q in z
order	Integer	Truncation order

Examples

```
q> heine3([(1,1,1), (1,1,2)], [(1,1,3)], 1, 1, 1, 10)
(prefactor, transformed_series)
```

Edge Cases and Constraints

- Requires exactly 2 upper and 1 lower parameter.
- The prefactor involves multiple infinite products and their quotients.

Related: heine1, heine2, phi

10.2 Advanced Transformations

Beyond the classical Heine transformations, two important higher-order transformations are available for balanced and very-well-poised series.

10.2.1 sears_transform

```
sears_transform(upper_list, lower_list, z_num, z_den, z_pow, order)
```

Apply Sears' transformation to a balanced terminating ${}_4\varphi_3$ series. Transforms one balanced ${}_4\varphi_3$ into an equivalent ${}_4\varphi_3$ with rearranged parameters. A series ${}_4\varphi_3$ is *balanced* when $qa_1a_2a_3a_4 = b_1b_2b_3z$ and one of the upper parameters is q^{-n} (terminating condition).

Parameters

Name	Type	Description
upper_list	List of (Int, Int, Int)	Four upper parameters as triples
lower_list	List of (Int, Int, Int)	Three lower parameters as triples
z_num	Integer	Numerator of z
z_den	Integer	Denominator of z

Name	Type	Description
z_pow	Integer	Power of q in z
order	Integer	Truncation order

Examples

```
q> sears_transform([(1,1,0),(1,1,1),(1,1,2),(1,1,-3)], [(1,1,3),(1,1,4),(1,1,5)], 1, 1, 0, 10)
(prefactor, transformed_4_phi_3)
```

Edge Cases and Constraints

- The series must satisfy the balanced condition; behavior is undefined otherwise.
- Requires exactly 4 upper and 3 lower parameters.

Related: watson_transform, phi, find_transformation_chain

10.2.2 watson_transform

```
watson_transform(upper_list, lower_list, z_num, z_den, z_pow, order)
```

Apply Watson's transformation to reduce a very-well-poised ${}_8\varphi_7$ to a balanced ${}_4\varphi_3$ via the Watson–Whipple identity. A series is *very-well-poised* when the upper parameters satisfy certain symmetry conditions involving $a_1^{1/2}$ and $-a_1^{1/2}$.

This is one of the deepest classical transformations and is the basis for many partition and Rogers–Ramanujan type identities.

Parameters

Name	Type	Description
upper_list	List of (Int, Int, Int)	Eight upper parameters as triples
lower_list	List of (Int, Int, Int)	Seven lower parameters as triples
z_num	Integer	Numerator of z
z_den	Integer	Denominator of z
z_pow	Integer	Power of q in z
order	Integer	Truncation order

Examples

```
q> watson_transform(upper, lower, 1, 1, 0, 10)
(prefactor, reduced_4_phi_3)
```

Edge Cases and Constraints

- The series must satisfy the very-well-poised condition.
- Requires exactly 8 upper and 7 lower parameters.

Related: sears_transform, phi, find_transformation_chain

10.2.3 find_transformation_chain

```
find_transformation_chain(src_upper, src_lower, src_z_n, src_z_d, src_z_p, tgt_upper,
tgt_lower, tgt_z_n, tgt_z_d, tgt_z_p, max_depth, order)
```

Search for a chain of Heine, Sears, and Watson transformations connecting a source hypergeometric series to a target, using breadth-first search (BFS) up to the specified maximum depth. At each step, all applicable transformations are tried, and the resulting series is compared against the target up to the truncation order. Returns the list of transformation steps if a path is found, or an empty list if no chain of length $\leq$ max_depth connects the two series.

Parameters

Name	Type	Description
src_upper	List of (Int, Int, Int)	Source series upper parameters
src_lower	List of (Int, Int, Int)	Source series lower parameters
src_z_n	Integer	Source z numerator
src_z_d	Integer	Source z denominator
src_z_p	Integer	Source z power of q
tgt_upper	List of (Int, Int, Int)	Target series upper parameters
tgt_lower	List of (Int, Int, Int)	Target series lower parameters
tgt_z_n	Integer	Target z numerator
tgt_z_d	Integer	Target z denominator
tgt_z_p	Integer	Target z power of q
max_depth	Integer	Maximum number of transformation steps
order	Integer	Truncation order for series comparison

Examples

```
q> find_transformation_chain([(1,1,1),(1,1,2)], [(1,1,3)], 1,1,1, [(1,1,2),(1,1,1)], [(1,1,3)], 1,1,1,
3, 10)
list of transformation steps (or empty if no path found)
```

Edge Cases and Constraints

- Returns an empty list if no chain of length $\leq$ max_depth connects the source to the target.
- Search time grows exponentially with `max_depth` since each step may branch into multiple transformations.
- The order parameter controls how many coefficients are compared – higher values give more confidence but cost more.

Related: heine1, heine2, heine3, sears_transform, watson_transform

11 Mock Theta Functions and Bailey Chains

Ramanujan introduced mock theta functions in his last letter to Hardy in January 1920. He listed 17 examples — seven of third order, ten of fifth order — and described them as functions at q-series level that "enter into mathematics as beautifully as the ordinary theta functions." Unlike classical theta functions, mock theta functions are *not* modular forms, but they exhibit a partial modularity that resisted precise characterization for over 80 years. The theory was finally unified by Zwegers (2002), who showed that every mock theta function is the holomorphic part of a harmonic Maass form, and that the Appell–Lerch sum provides the natural framework for this correspondence.

Bailey chains provide a complementary algebraic approach to q-series identities. A Bailey pair relative to a is a pair of sequences (α_n, β_n) satisfying a linear relation involving q-Pochhammer products. Bailey's lemma transforms one Bailey pair into another, and iterating this transformation produces a *Bailey chain* — an infinite sequence of increasingly complex identities, including the Rogers–Ramanujan identities as special cases.

11.1 Third-Order Mock Theta Functions

Ramanujan's original seven third-order mock theta functions from his 1920 letter to Hardy. Each takes a single `order` parameter specifying the truncation order of the resulting q-series.

11.1.1 mock_theta_f3

```
mock_theta_f3(order)
```

Compute Ramanujan's third-order mock theta function $f(q)$. This is perhaps the most studied of all mock theta functions, appearing in Ramanujan's original letter and in numerous subsequent identities.

Mathematical Definition

$$f(q) = \sum_{n \geq 0} \frac{q^{n^2}}{(-q;q)_n^2}$$

Parameters

Name	Type	Description
order	Integer	Truncation order for the output series

Examples

```
q> mock_theta_f3(10)
1 + q - 2*q^2 + 3*q^3 + ... + O(q^10)
```

```
q> mock_theta_f3(5)
1 + q - 2*q^2 + 3*q^3 - 3*q^4 + O(q^5)
```

Edge Cases and Constraints

- The order must be a positive integer.
- At low orders, the series may appear polynomial; the mock theta property manifests in the asymptotic behavior of coefficients.

Related: mock_theta_phi3, mock_theta_psi3, mock_theta_chi3, mock_theta_omega3, mock_theta_nu3, mock_theta_rho3

11.1.2 mock_theta_phi3

```
mock_theta_phi3(order)
```

Compute Ramanujan's third-order mock theta function $\varphi(q)$.

Mathematical Definition

$$\varphi(q) = \sum_{n \geq 0} \frac{q^{n^2}}{(-q^2; q^2)_n}$$

Parameters

Name	Type	Description
order	Integer	Truncation order for the output series

Examples

```
q> mock_theta_phi3(10)
1 + q + q^2 + ... + O(q^10)
```

Edge Cases and Constraints

- The order must be a positive integer.

Related: mock_theta_f3, mock_theta_psi3, mock_theta_chi3

11.1.3 mock_theta_psi3

```
mock_theta_psi3(order)
```

Compute Ramanujan's third-order mock theta function $\psi(q)$. Note that the summation starts at $n = 1$, so the constant term is zero.

Mathematical Definition

$$\psi(q) = \sum_{n \geq 1} \frac{q^{n^2}}{(q;q^2)_n}$$

Parameters

Name	Type	Description
order	Integer	Truncation order for the output series

Examples

```
q> mock_theta_psi3(10)
q + q^2 + ... + O(q^10)
```

Edge Cases and Constraints

- The order must be a positive integer.
- The constant term is always zero because the sum starts at $n = 1$.

Related: mock_theta_f3, mock_theta_phi3, mock_theta_chi3

11.1.4 mock_theta_chi3

```
mock_theta_chi3(order)
```

Compute Ramanujan's third-order mock theta function $\chi(q)$.

Mathematical Definition

$$\chi(q) = \sum_{n \geq 0} \frac{q^{n^2}(-q;q)_n}{\prod_{k \geq 1}(1 - q^k + q^{2k})}$$

Parameters

Name	Type	Description
order	Integer	Truncation order for the output series

Examples

```
q> mock_theta_chi3(10)
1 + q + ... + O(q^10)
```

Edge Cases and Constraints

- The order must be a positive integer.
- The denominator product involves cubic roots of unity factors $1 - q^k + q^{2k}$.

Related: mock_theta_f3, mock_theta_phi3, mock_theta_psi3

11.1.5 mock_theta_omega3

```
mock_theta_omega3(order)
```

Compute Ramanujan's third-order mock theta function $\omega(q)$.

Mathematical Definition

$$\omega(q) = \sum_{n \geq 0} \frac{q^{2n(n+1)}}{(q;q^2)_{n+1}^2}$$

Parameters

Name	Type	Description
order	Integer	Truncation order for the output series

Examples

```
q> mock_theta_omega3(10)
1 + 2*q^2 + ... + O(q^10)
```

Edge Cases and Constraints

- The order must be a positive integer.
- The exponent $2n(n+1)$ grows quadratically, so few terms contribute at low orders.

Related: mock_theta_nu3, mock_theta_rho3, mock_theta_f3

11.1.6 mock_theta_nu3

```
mock_theta_nu3(order)
```

Compute Ramanujan's third-order mock theta function $\nu(q)$.

Mathematical Definition

$$\nu(q) = \sum_{n \geq 0} \frac{(-1)^n q^{n(n+1)}}{(-q;q^2)_{n+1}}$$

Parameters

Name	Type	Description
order	Integer	Truncation order for the output series

Examples

```
q> mock_theta_nu3(10)
1 - q + ... + O(q^10)
```

Edge Cases and Constraints

- The order must be a positive integer.
- The alternating sign $(-1)^n$ causes sign changes in the coefficients.

Related: mock_theta_omega3, mock_theta_rho3, mock_theta_f3

11.1.7 mock_theta_rho3

```
mock_theta_rho3(order)
```

Compute the third-order mock theta function $\rho(q)$.

Mathematical Definition

$$\rho(q) = \sum_{n \geq 0} \frac{q^{2n(n+1)}}{\prod_{m=1}^{n+1}(1 + q^m + q^{2m})}$$

Parameters

Name	Type	Description
order	Integer	Truncation order for the output series

Examples

```
q> mock_theta_rho3(10)
1 + q^2 + ... + O(q^10)
```

Edge Cases and Constraints

- The order must be a positive integer.
- Like $\chi(q)$, the denominator involves cubic-root-of-unity factors.

Related: mock_theta_omega3, mock_theta_nu3, mock_theta_chi3

11.2 Fifth-Order Mock Theta Functions

Ramanujan's ten fifth-order mock theta functions, also from his last letter to Hardy. These are organized into five pairs: (f_0, f_1), (F_0, F_1), (φ_0, φ_1), (ψ_0, ψ_1), and (χ_0, χ_1). Each takes a single `order` parameter.

11.2.1 mock_theta_f0_5

```
mock_theta_f0_5(order)
```

Compute Ramanujan's fifth-order mock theta function $f_0(q)$.

Mathematical Definition

$$f_0(q) = \sum_{n \geq 0} \frac{q^{n^2}}{(-q;q)_n}$$

Parameters

Name	Type	Description
order	Integer	Truncation order for the output series

Examples

```
q> mock_theta_f0_5(10)
1 + q + ... + O(q^10)
```

Edge Cases and Constraints

- The order must be a positive integer.

Related: mock_theta_f1_5, mock_theta_f3

11.2.2 mock_theta_f1_5

```
mock_theta_f1_5(order)
```

Compute Ramanujan's fifth-order mock theta function $f_1(q)$. Note that the summation starts at $n = 1$.

Mathematical Definition

$$f_1(q) = \sum_{n \geq 1} \frac{q^{n^2}}{(q;q)_n}$$

Parameters

Name	Type	Description
order	Integer	Truncation order for the output series

Examples

```
q> mock_theta_f1_5(10)
q + q^2 + ... + O(q^10)
```

Edge Cases and Constraints

- The order must be a positive integer.
- The constant term is zero because the sum starts at $n = 1$.

Related: mock_theta_f0_5, mock_theta_f3

11.2.3 mock_theta_cap_f0_5

```
mock_theta_cap_f0_5(order)
```

Compute Ramanujan's fifth-order mock theta function $F_0(q)$ (capital F).

Mathematical Definition

$$F_0(q) = \sum_{n \geq 0} \frac{q^{2n^2}}{(q;q^2)_n}$$

Parameters

Name	Type	Description
order	Integer	Truncation order for the output series

Examples

```
q> mock_theta_cap_f0_5(10)
1 + q^2 + ... + O(q^10)
```

Edge Cases and Constraints

- The order must be a positive integer.

Related: mock_theta_cap_f1_5, mock_theta_f0_5

11.2.4 mock_theta_cap_f1_5

```
mock_theta_cap_f1_5(order)
```

Compute Ramanujan's fifth-order mock theta function $F_1(q)$ (capital F).

Mathematical Definition

$$F_1(q) = \sum_{n \geq 1} \frac{q^{2n^2-2n+1}}{(q;q^2)_n}$$

Parameters

Name	Type	Description
order	Integer	Truncation order for the output series

Examples

```
q> mock_theta_cap_f1_5(10)
q + q^3 + ... + O(q^10)
```

Edge Cases and Constraints

- The order must be a positive integer.
- The exponent $2n^2 - 2n + 1$ ensures the leading term is q.

Related: mock_theta_cap_f0_5, mock_theta_f1_5

11.2.5 mock_theta_phi0_5

```
mock_theta_phi0_5(order)
```

Compute Ramanujan's fifth-order mock theta function $\varphi_0(q)$.

Mathematical Definition

$$\varphi_0(q) = \sum_{n \geq 0} q^{n^2} (-q;q^2)_n$$

Parameters

Name	Type	Description
order	Integer	Truncation order for the output series

Examples

```
q> mock_theta_phi0_5(10)
1 + q + ... + O(q^10)
```

Edge Cases and Constraints

- The order must be a positive integer.
- Unlike most mock theta functions, the q-Pochhammer factor appears in the numerator.

Related: mock_theta_phi1_5, mock_theta_phi3

11.2.6 mock_theta_phi1_5

```
mock_theta_phi1_5(order)
```

Compute Ramanujan's fifth-order mock theta function $\varphi_1(q)$.

Mathematical Definition

$$\varphi_1(q) = \sum_{n\geq 0} q^{(n+1)^2} (-q;q^2)_n$$

Parameters

Name	Type	Description
order	Integer	Truncation order for the output series

Examples

```
q> mock_theta_phi1_5(10)
q + q^2 + ... + O(q^10)
```

Edge Cases and Constraints

- The order must be a positive integer.

Related: mock_theta_phi0_5, mock_theta_phi3

11.2.7 mock_theta_psi0_5

```
mock_theta_psi0_5(order)
```

Compute Ramanujan's fifth-order mock theta function $\psi_0(q)$.

Mathematical Definition

$$\psi_0(q) = \sum_{n\geq 0} q^{(n+1)(n+2)/2} (-q;q)_n$$

Parameters

Name	Type	Description
order	Integer	Truncation order for the output series

Examples

```
q> mock_theta_psi0_5(10)
q + q^2 + ... + O(q^10)
```

Edge Cases and Constraints

- The order must be a positive integer.
- The exponent $(n+1)\frac{n+2}{2}$ produces triangular-number-like powers.

Related: mock_theta_psi1_5, mock_theta_psi3

11.2.8 mock_theta_psi1_5

```
mock_theta_psi1_5(order)
```

Compute Ramanujan's fifth-order mock theta function $\psi_1(q)$. Note that the summation starts at $n = 1$.

Mathematical Definition

$$\psi_1(q) = \sum_{n \geq 1} q^{n(n+1)/2} (-q;q)_{n-1}$$

Parameters

Name	Type	Description
order	Integer	Truncation order for the output series

Examples

```
q> mock_theta_psi1_5(10)
q + q^2 + ... + O(q^10)
```

Edge Cases and Constraints

- The order must be a positive integer.
- The q-Pochhammer factor uses index $n - 1$.

Related: mock_theta_psi0_5, mock_theta_psi3

11.2.9 mock_theta_chi0_5

```
mock_theta_chi0_5(order)
```

Compute Ramanujan's fifth-order mock theta function $\chi_0(q)$. This function is computed internally via a $q \to -q$ composition technique.

Mathematical Definition

$$\chi_0(q) = \sum_{n \geq 0} \frac{q^n (-q;q)_{n-1}}{(q^{n+1};q)_n}$$

Parameters

Name	Type	Description
order	Integer	Truncation order for the output series

Examples

```
q> mock_theta_chi0_5(10)
1 + q + ... + O(q^10)
```

Edge Cases and Constraints

- The order must be a positive integer.
- Uses the $q \to -q$ composition method internally for numerical stability.

Related: mock_theta_chi1_5, mock_theta_chi3

11.2.10 mock_theta_chi1_5

```
mock_theta_chi1_5(order)
```

Compute Ramanujan's fifth-order mock theta function $\chi_1(q)$. Like χ_0, this function uses the $q \to -q$ composition technique.

Mathematical Definition

$$\chi_1(q) = \sum_{n \geq 0} \frac{q^n (-q;q)_n}{(q^{n+1};q)_n}$$

Parameters

Name	Type	Description
order	Integer	Truncation order for the output series

Examples

```
q> mock_theta_chi1_5(10)
q + q^2 + ... + O(q^10)
```

Edge Cases and Constraints

- The order must be a positive integer.
- Uses the $q \to -q$ composition method internally for numerical stability.

Related: mock_theta_chi0_5, mock_theta_chi3

11.3 Seventh-Order Mock Theta Functions

Three seventh-order mock theta functions, discovered later than Ramanujan's original third- and fifth-order examples. Each takes a single `order` parameter.

11.3.1 mock_theta_cap_f0_7

```
mock_theta_cap_f0_7(order)
```

Compute the seventh-order mock theta function $F_0(q)$.

Mathematical Definition

$$F_0(q) = \sum_{n \geq 0} \frac{q^{n^2}}{(q^{n+1}; q)_n}$$

Parameters

Name	Type	Description
order	Integer	Truncation order for the output series

Examples

```
q> mock_theta_cap_f0_7(10)
1 + q + ... + O(q^10)
```

Edge Cases and Constraints

- The order must be a positive integer.

Related: mock_theta_cap_f1_7, mock_theta_cap_f2_7

11.3.2 mock_theta_cap_f1_7

```
mock_theta_cap_f1_7(order)
```

Compute the seventh-order mock theta function $F_1(q)$. Note that the summation starts at $n = 1$.

Mathematical Definition

$$F_1(q) = \sum_{n \geq 1} \frac{q^{n^2}}{(q^n; q)_n}$$

Parameters

Name	Type	Description
order	Integer	Truncation order for the output series

Examples

```
q> mock_theta_cap_f1_7(10)
q + q^2 + ... + O(q^10)
```

Edge Cases and Constraints

- The order must be a positive integer.
- The constant term is zero because the sum starts at $n = 1$.

Related: mock_theta_cap_f0_7, mock_theta_cap_f2_7

11.3.3 mock_theta_cap_f2_7

```
mock_theta_cap_f2_7(order)
```

Compute the seventh-order mock theta function $F_2(q)$.

Mathematical Definition

$$F_2(q) = \sum_{n \geq 0} \frac{q^{n(n+1)}}{(q^{n+1}; q)_{n+1}}$$

Parameters

Name	Type	Description
order	Integer	Truncation order for the output series

Examples

```
q> mock_theta_cap_f2_7(10)
1 + q + ... + O(q^10)
```

Edge Cases and Constraints

- The order must be a positive integer.

Related: mock_theta_cap_f0_7, mock_theta_cap_f1_7

11.4 Appell–Lerch Sums

Zwegers' unifying framework for mock theta functions. The Appell–Lerch sum $m(a, z, q)$ and the universal mock theta functions g_2 and g_3 provide a systematic way to express all classical mock theta functions and to understand their modular transformation properties.

11.4.1 appell_lerch_m

```
appell_lerch_m(a_pow, z_pow, order)
```

Compute the Appell–Lerch sum $m(a, z, q)$ where $a = q^{a_{\text{pow}}}$ and $z = q^{z_{\text{pow}}}$. This function provides the foundational building block for Zwegers' theory of mock theta functions.

Mathematical Definition

$$m(a, z, q) = \frac{1}{j(z; q)} \sum_{r \in \mathbb{Z}} \frac{(-1)^r z^r q^{r(r-1)/2}}{1 - aq^r z}$$

where $j(z; q) = (z; q)_\infty (q/z; q)_\infty (q; q)_\infty$ is the Jacobi theta function normalization.

Parameters

Name	Type	Description
a_pow	Integer	Exponent for $a = q^{a_{\text{pow}}}$
z_pow	Integer	Exponent for $z = q^{z_{\text{pow}}}$
order	Integer	Truncation order for the output series

Examples

```
q> appell_lerch_m(1, 1, 10)
series in q truncated to order 10
```

```
q> appell_lerch_m(2, 1, 15)
series in q truncated to order 15
```

Edge Cases and Constraints

- The parameters `a_pow` and `z_pow` must be integers.
- When $z = q^0 = 1$ or $a = q^0 = 1$, the sum may have poles; the implementation handles cancellation carefully.
- The Jacobi theta normalization $j(z;q)$ in the denominator is computed as a triple product.

Related: universal_mock_theta_g2, universal_mock_theta_g3, mock_theta_f3

11.4.2 universal_mock_theta_g2

```
universal_mock_theta_g2(a_pow, order)
```

Compute the universal mock theta function $g_2(a;q)$ where $a = q^{a_{\text{pow}}}$. The function g_2 relates to the Appell–Lerch sum via algebraic identities and provides a second-order universal mock theta function.

Mathematical Definition

$$g_2(a;q) \quad \text{where } a = q^{a_{\text{pow}}}$$

Parameters

Name	Type	Description
a_pow	Integer	Exponent for $a = q^{a_{\text{pow}}}$
order	Integer	Truncation order for the output series

Examples

```
q> universal_mock_theta_g2(1, 10)
series in q truncated to order 10
```

Edge Cases and Constraints

- The parameter `a_pow` must be an integer.
- Related to the Appell–Lerch sum $m(a,z,q)$ through specialization of parameters.

Related: universal_mock_theta_g3, appell_lerch_m

11.4.3 universal_mock_theta_g3

```
universal_mock_theta_g3(a_pow, order)
```

Compute the universal mock theta function $g_3(a;q)$ where $a = q^{a_{\text{pow}}}$. The function g_3 relates to the Appell–Lerch sum via algebraic identities and provides a third-order universal mock theta function.

Mathematical Definition

$$g_3(a;q) \quad \text{where } a = q^{a_{\text{pow}}}$$

Parameters

Name	Type	Description
a_pow	Integer	Exponent for $a = q^{a_{\text{pow}}}$
order	Integer	Truncation order for the output series

Examples

```
q> universal_mock_theta_g3(1, 10)
series in q truncated to order 10
```

Edge Cases and Constraints

- The parameter `a_pow` must be an integer.
- Related to the Appell–Lerch sum $m(a, z, q)$ through specialization of parameters.

Related: universal_mock_theta_g2, appell_lerch_m

11.5 Bailey Chains

A *Bailey pair* relative to a is a pair of sequences (α_n, β_n) satisfying

$$\beta_n = \sum_{k=0}^{n} \frac{\alpha_k}{(q;q)_{n-k}(aq;q)_{n+k}}.$$

Bailey's lemma produces a new Bailey pair from an existing one by applying a transformation matrix involving additional parameters b and c. Iterating this process produces a *Bailey chain*: an infinite sequence of pairs, each yielding a new q-series identity. The Rogers–Ramanujan identities emerge as a special case at depth 1 starting from the unit Bailey pair.

11.5.1 bailey_weak_lemma

```
bailey_weak_lemma(pair_code, a_num, a_den, a_pow, max_n, order)
```

Apply the weak form of Bailey's lemma to a known Bailey pair. The weak lemma is the simplest form of the Bailey transformation, without the additional parameters b and c of the full lemma.

Parameters

Name	Type	Description
pair_code	Integer	Selects the initial Bailey pair: 0 = Unit, 1 = Rogers–Ramanujan, 2 = q-Binomial
a_num	Integer	Numerator of the parameter $a = (a_{\text{num}}/a_{\text{den}}) \cdot q^{a_{\text{pow}}}$

Name	Type	Description
a_den	Integer	Denominator of the parameter a
a_pow	Integer	Power of q in the parameter a
max_n	Integer	Maximum index for the Bailey pair computation
order	Integer	Truncation order for the output series

Examples

```
q> bailey_weak_lemma(1, 1, 1, 0, 10, 20)
(alpha_series, beta_series)
```

```
q> bailey_weak_lemma(0, 1, 1, 0, 5, 15)
(alpha_series, beta_series)
```

Edge Cases and Constraints

- The `pair_code` must be 0, 1, or 2.
- The denominator `a_den` must be nonzero.
- The `max_n` parameter controls how many terms of the Bailey pair are computed.

Related: bailey_apply_lemma, bailey_chain, bailey_discover

11.5.2 bailey_apply_lemma

```
bailey_apply_lemma(pair_code, a_n, a_d, a_p, b_n, b_d, b_p, c_n, c_d, c_p, max_n, order)
```

Apply the full form of Bailey's lemma to a Bailey pair, producing a new pair. The parameters b and c control the transformation matrix, allowing more general identity generation than the weak lemma.

Parameters

Name	Type	Description
pair_code	Integer	Selects the initial Bailey pair: 0 = Unit, 1 = Rogers–Ramanujan, 2 = q-Binomial
a_n, a_d, a_p	Integer	Parameter $a = (a_n/a_d) \cdot q^{a_p}$
b_n, b_d, b_p	Integer	Parameter $b = (b_n/b_d) \cdot q^{b_p}$
c_n, c_d, c_p	Integer	Parameter $c = (c_n/c_d) \cdot q^{c_p}$
max_n	Integer	Maximum index for the Bailey pair computation
order	Integer	Truncation order for the output series

Examples

```
q> bailey_apply_lemma(0, 1,1,0, 1,1,1, 1,1,2, 10, 20)
(new_alpha_series, new_beta_series)
```

Edge Cases and Constraints

- All denominators (`a_d`, `b_d`, `c_d`) must be nonzero.
- The `pair_code` must be 0, 1, or 2.
- The full lemma has 12 parameters; ensure all are provided.

Related: bailey_weak_lemma, bailey_chain, bailey_discover

11.5.3 bailey_chain

```
bailey_chain(pair_code, a_n, a_d, a_p, b_n, b_d, b_p, c_n, c_d, c_p, depth, max_n, order)
```

Iterate the Bailey chain to the specified depth, starting from a known Bailey pair. Each iteration applies Bailey's lemma to produce a new pair. At depth 1 with the Rogers–Ramanujan pair, this yields the classical Rogers–Ramanujan identities.

Parameters

Name	Type	Description
pair_code	Integer	Selects the initial Bailey pair: 0 = Unit, 1 = Rogers–Ramanujan, 2 = q-Binomial
a_n, a_d, a_p	Integer	Parameter a (rational times q-power)
b_n, b_d, b_p	Integer	Parameter b (rational times q-power)
c_n, c_d, c_p	Integer	Parameter c (rational times q-power)
depth	Integer	Number of Bailey lemma iterations to perform
max_n	Integer	Maximum index for the Bailey pair computation
order	Integer	Truncation order for the output series

Examples

```
q> bailey_chain(1, 1,1,0, 1,1,1, 1,1,2, 3, 10, 20)
(alpha_at_depth, beta_at_depth)
```

Edge Cases and Constraints

- The `depth` must be a non-negative integer; depth 0 returns the original pair.
- Higher depths produce increasingly complex series; truncation order should be generous.
- This function has 13 parameters.

Related: bailey_weak_lemma, bailey_apply_lemma, bailey_discover

11.5.4 bailey_discover

```
bailey_discover(lhs, rhs, a_num, a_den, a_pow, max_depth, order)
```

Discover a Bailey pair that proves the identity `lhs` = `rhs`. The algorithm tries multiple strategies in order: trivial equality check, database lookup against known Bailey pairs, weak lemma matching, and iterative chain depth search up to `max_depth`.

Parameters

Name	Type	Description
lhs	Series	Left-hand side of the identity to prove
rhs	Series	Right-hand side of the identity to prove
a_num	Integer	Numerator of the parameter a
a_den	Integer	Denominator of the parameter a
a_pow	Integer	Power of q in the parameter a
max_depth	Integer	Maximum Bailey chain depth to search
order	Integer	Truncation order for series comparison

Examples

```
q> bailey_discover(lhs_series, rhs_series, 1, 1, 0, 3, 20)
proof description (or None if not found)
```

Edge Cases and Constraints

- Both `lhs` and `rhs` must be q-series (not integers or other types).
- Higher `max_depth` values increase search time exponentially.
- Returns `None` if no Bailey pair proof is found within the search depth.

Related: bailey_chain, bailey_weak_lemma, bailey_apply_lemma, prove_eta_id

12 Identity Proving

Algorithmic methods for proving q-series identities. These functions go beyond empirical verification — checking that coefficients of two series agree to some finite order — to provide rigorous mathematical proofs. The eta-quotient prover uses the valence formula for modular forms: if a modular form of weight k and level N has more zeros (counted with multiplicity at cusps) than $kN/12 \cdot \prod_{p \mid N}(1 + 1/p)$, then it is identically zero. The q-Gosper, q-Zeilberger, and WZ machinery provides computer-algebra proofs of hypergeometric summation identities through creative telescoping and certificate verification.

The `search_identities` function provides a lookup interface into the built-in identity database, while `prove_nonterminating` extends proving capabilities to nonterminating hypergeometric identities (available only through the Python API).

12.0.1 prove_eta_id

```
prove_eta_id(terms_list, level)
```

Prove an eta-quotient identity using the valence formula for modular forms. The function verifies that a linear combination of eta quotients is identically zero by checking that the number of zeros exceeds the bound imposed by the valence formula.

Mathematical Definition

Uses the *valence formula* for modular forms: if f is a modular form of weight k for $\Gamma_0(N)$ with $\text{ord}(f) > k\frac{N}{12}\prod_{p \mid N}\left(1 + \frac{1}{p}\right)$, then $f \equiv 0$.

Each term in `terms_list` specifies an eta quotient $\prod \eta(d \cdot \tau)^{r_d}$ together with a scalar coefficient.

Parameters

Name	Type	Description
terms_list	List	List of `(eta_args, coefficient)` pairs, where `eta_args` specifies the eta quotient via base-exponent pairs
level	Integer	Modular group level N for $\Gamma_0(N)$

Examples

```
q> prove_eta_id([([1,1], 1), ([1,-1], -1)], 1)
true (identity proven) or false
```

```
q> prove_eta_id([([1,24], 1), ([2,-24], -1)], 2)
true
```

Edge Cases and Constraints

- The `level` must divide the LCM of all eta bases appearing in the terms.
- Works for modular forms of weight 0 (eta quotients whose exponents sum to zero).
- Returns `false` if the coefficient check is insufficient to prove the identity, which does not necessarily mean the identity is false.
- Large levels increase the valence bound and require more coefficients to be checked.

Related: etaq, search_identities

12.0.2 search_identities

```
search_identities(search_type)
```

Search for identities of a given type in the built-in database. Returns a list of known identities matching the search criteria. This is useful for finding starting points for identity exploration or for checking whether a suspected identity is already known.

Parameters

Name	Type	Description
search_type	String	Type of identity to search for: `"theta"`, `"eta"`, `"mock"`, `"bailey"`, or `"product"`

Examples

```
q> search_identities("theta")
list of matching theta-function identities
```

```
q> search_identities("bailey")
list of matching Bailey pair identities
```

Edge Cases and Constraints

- The `search_type` must be a string (enclosed in double quotes in the REPL).
- An unrecognized search type returns an empty list.
- The database is curated and does not contain all known identities; absence does not imply non-existence.

Related: prove_eta_id, bailey_discover

12.0.3 q_gosper

```
q_gosper(upper_list, lower_list, z_num, z_den, z_pow, q_num, q_den)
```

Apply the q-Gosper algorithm for indefinite q-hypergeometric summation. Given a q-hypergeometric term t_k specified by its term ratio t_{k+1}/t_k, the algorithm searches for a q-rational function y_k such that the sum $\sum t_k$ has the closed-form antidifference $y_k t_k$.

Mathematical Definition

The algorithm finds y_k (a q-rational function of q^k) such that

$$\sum_{k=0}^{n} t_k = y_{n+1} t_{n+1} - y_0 t_0$$

where the term ratio t_{k+1}/t_k is determined by the upper and lower parameter lists. Not all q-hypergeometric sums are Gosper-summable; the algorithm returns `None` when no q-hypergeometric antidifference exists.

Parameters

Name	Type	Description
upper_list	List	Upper parameters as `(num, den, pow)` triples specifying $q^{\text{pow}} \cdot \frac{\text{num}}{\text{den}}$
lower_list	List	Lower parameters as `(num, den, pow)` triples
z_num	Integer	Numerator of the argument z
z_den	Integer	Denominator of the argument z
z_pow	Integer	Power of q in z
q_num	Integer	Numerator of the base q (usually 1)
q_den	Integer	Denominator of the base q (usually 1)

Examples

```
q> q_gosper([(1,1,0)], [(1,1,1)], 1, 1, 0, 1, 1)
closed-form antidifference (or None)
```

Edge Cases and Constraints

- Not all q-hypergeometric sums have q-hypergeometric antidifferences.
- Returns `None` if no Gosper-summable form exists.
- The `q_num` and `q_den` parameters allow working with non-standard bases.

Related: q_zeilberger, phi, verify_wz

12.0.4 q_zeilberger

```
q_zeilberger(upper_list, lower_list, z_num, z_den, z_pow, n, q_num, q_den, max_order)
```

Apply q-Zeilberger's creative telescoping algorithm to find a recurrence relation for a definite q-hypergeometric sum. Given a summand $F(n,k)$ specified by the upper/lower parameter lists, the algorithm produces a recurrence $\sum_{j=0}^{J} a_{j(q^n)} S(n+j) = 0$ for the definite sum $S(n) = \sum_k F(n,k)$.

Mathematical Definition

Creative telescoping finds polynomials $a_0(q^n), ..., a_{J(q^n)}$ and a *WZ certificate* $R(n,k)$ such that

$$\sum_{j=0}^{J} a_{j(q^n)} F(n+j,k) = G(n,k+1) - G(n,k)$$

where $G(n,k) = R(n,k)F(n,k)$. Summing over k yields the recurrence for $S(n)$.

Parameters

Name	Type	Description
upper_list	List	Upper parameters as `(num, den, pow)` triples
lower_list	List	Lower parameters as `(num, den, pow)` triples
z_num	Integer	Numerator of the argument z
z_den	Integer	Denominator of the argument z
z_pow	Integer	Power of q in z
n	Integer	Evaluation point for the summation index
q_num	Integer	Numerator of the base q
q_den	Integer	Denominator of the base q
max_order	Integer	Maximum recurrence order to search

Examples

```
q> q_zeilberger([(1,1,0),(1,1,1)], [(1,1,2)], 1, 1, 0, 5, 1, 1, 3)
(recurrence_coeffs, certificate)
```

Edge Cases and Constraints

- The `max_order` controls the search depth; higher values find higher-order recurrences but take longer.
- Returns `None` if no recurrence of order $\leq$ max_order exists.
- The returned certificate can be verified independently using `verify_wz`.

Related: verify_wz, q_gosper, q_petkovsek

12.0.5 verify_wz

```
verify_wz(upper_list, lower_list, z_num, z_den, z_pow, n, q_num, q_den, max_order,
max_k)
```

Verify a Wilf–Zeilberger proof certificate. First runs `q_zeilberger` to produce the recurrence and certificate, then verifies that the certificate satisfies the WZ pair equations. A successful verification constitutes a rigorous proof of the underlying summation identity.

Parameters

Name	Type	Description
upper_list	List	Upper parameters as `(num, den, pow)` triples
lower_list	List	Lower parameters as `(num, den, pow)` triples
z_num	Integer	Numerator of the argument z
z_den	Integer	Denominator of the argument z
z_pow	Integer	Power of q in z
n	Integer	Evaluation point for the summation index
q_num	Integer	Numerator of the base q
q_den	Integer	Denominator of the base q
max_order	Integer	Maximum recurrence order to search
max_k	Integer	Maximum k value for certificate verification

Examples

```
q> verify_wz([(1,1,0),(1,1,1)], [(1,1,2)], 1, 1, 0, 5, 1, 1, 3, 10)
true (certificate valid) or false
```

Edge Cases and Constraints

- Returns `false` if `q_zeilberger` fails to find a recurrence.
- The `max_k` parameter limits the range over which the certificate is checked; larger values give stronger verification.
- A `true` result constitutes a rigorous proof, not merely empirical evidence.

Related: q_zeilberger, q_gosper

12.0.6 q_petkovsek

```
q_petkovsek(coeff_list, q_num, q_den)
```

Solve a q-holonomic recurrence using the q-Petkovsek algorithm. Given a recurrence with polynomial coefficients in q^n, the algorithm finds all solutions that are q-hypergeometric terms — that is, sequences a_n where a_{n+1}/a_n is a rational function of q^n.

Parameters

Name	Type	Description
coeff_list	List	Polynomial coefficients of the recurrence, from lowest to highest order
q_num	Integer	Numerator of the base q
q_den	Integer	Denominator of the base q

Examples

```
q> q_petkovsek([1, -1, 1], 1, 1)
list of q-hypergeometric solutions
```

Edge Cases and Constraints

- The `coeff_list` must contain at least 2 elements (a nontrivial recurrence).
- Returns an empty list if no q-hypergeometric solutions exist.
- The algorithm finds *all* q-hypergeometric solutions, not just one.

Related: q_zeilberger, q_gosper

12.0.7 prove_nonterminating

```
prove_nonterminating(requires Python API)
```

Prove a nonterminating hypergeometric identity using symbolic parameter manipulation. This function requires closure support for representing symbolic parameters, which is available only through the Python API (`q_kangaroo` package), not the CLI REPL.

Examples

```
q> prove_nonterminating(...)
Error: prove_nonterminating requires the Python API
```

Edge Cases and Constraints

- **This function is NOT available in the CLI.** Calling it from the REPL produces an error message directing the user to the Python API.
- Requires the `q_kangaroo` Python package to be installed (via `pip install q_kangaroo`).
- Uses closure-based symbolic parameters to manipulate nonterminating series, which cannot be represented in the CLI's expression language.

Related: phi, try_summation, q_gosper

13 Worked Examples

This chapter demonstrates q-Kangaroo through extended examples that span multiple function groups. Each example shows a complete research workflow from problem statement through computational verification, with mathematical context and references to the source literature.

13.1 Euler's Pentagonal Theorem

Mathematical context. Euler's pentagonal number theorem (1750) states that the Euler function factors as

$$(q;q)_\infty = \prod_{k=1}^{\infty}(1-q^k) = \sum_{n=-\infty}^{\infty}(-1)^n q^{n(3n-1)/2}$$

The exponents $0, 1, 2, 5, 7, 12, 15, 22, \ldots$ are the *generalised pentagonal numbers* $\omega(n) = \frac{n(3n-1)}{2}$ for $n = 0, \pm 1, \pm 2, \ldots$ The identity is equivalent to the statement that the partition function $p(n)$ satisfies the recurrence $p(n) = p(n-1) + p(n-2) - p(n-5) - p(n-7) + \ldots$

Reference: Euler (1750); see Andrews, *The Theory of Partitions* (Addison-Wesley, 1976), Chapter 1.

13.1.1 REPL Workflow

Step 1. Compute $(q;q)_\infty$ to 20 terms using `aqprod`:

```
q> aqprod(1, 1, 1, infinity, 20)
1 - q - q^2 + q^5 + q^7 - q^12 - q^15 + O(q^20)
```

Observe the nonzero coefficients at $q^0, q^1, q^2, q^5, q^7, q^{12}, q^{15}$ — exactly the pentagonal numbers — and that the signs alternate in pairs $(+, -, -, +, +, -, -, \ldots)$.

Step 2. Confirm that the product is a single-factor eta quotient using `prodmake`:

```
q> prodmake(%, 10)
{1: 1, 2: 1, 3: 1, 4: 1, 5: 1, 6: 1, 7: 1, 8: 1, 9: 1, 10: 1}
```

Every factor $(1-q^k)$ appears with exponent 1, confirming $(q;q)_\infty = \prod_{k=1}^{\infty}(1-q^k)$.

Step 3. Verify via the eta representation:

```
q> etamake(aqprod(1, 1, 1, infinity, 50), 10)
{1: 1}
```

The result `{1: 1}` means exactly $\eta(\tau)$, the Dedekind eta function with $q = e^{2\pi i\tau}$.

Step 4. Cross-check: multiply by the partition generating function and confirm the result is 1:

```
q> f := aqprod(1, 1, 1, infinity, 20):
q> g := partition_gf(20):
q> f * g
1 + O(q^20)
```

The identity $(q;q)_\infty \cdot 1/(q;q)_\infty = 1$ is confirmed to 20 terms.

Takeaway. The signs and exponents in $(q;q)_\infty$ encode the pentagonal numbers. The `prodmake` and `etamake` functions verify the product structure directly, while multiplication by the partition generating function provides an independent check.

13.2 Ramanujan's Partition Congruences

Mathematical context. Ramanujan (1919) discovered that the partition function satisfies three remarkable congruences:

$$p(5n+4) \equiv 0 \pmod 5$$
$$p(7n+5) \equiv 0 \pmod 7$$
$$p(11n+6) \equiv 0 \pmod{11}$$

These are the only congruences of the form $p(\ell n + \delta) \equiv 0 \pmod \ell$ for primes $\ell \leq 31$. Proving these congruences motivated much of the theory of modular forms as applied to partitions.

Reference: Ramanujan (1919), "Some Properties of $p(n)$", *Proc. Cambridge Phil. Soc.* 19, 207–210. See also Berndt, *Ramanujan's Notebooks*, Vol. III (Springer, 1991).

13.2.1 REPL Workflow

Step 1. Generate the partition function to high order so that sifting produces enough terms for pattern recognition:

```
q> f := partition_gf(200):
```

Step 2. Use `findcong` to automatically discover congruences modulo 5, 7, and 11:

```
q> findcong(f, [5, 7, 11])
[[5, 4, 5], [7, 5, 7], [11, 6, 11]]
```

Each triple $[m, j, d]$ means $p(mn + j) \equiv 0 \pmod d$. The three Ramanujan congruences are recovered automatically.

Step 3. Verify the mod-5 congruence manually with `sift`. Extract the subsequence $p(5n+4)$ and observe that every coefficient is divisible by 5:

```
q> sift(f, 5, 4)
5 + 30*q + 135*q^2 + 490*q^3 + 1575*q^4 + 4565*q^5 + 12310*q^6 + 31185*q^7 + 75175*q^8 + 173525*q^9
+ O(q^10)
```

Coefficients: $5, 30, 135, 490, 1575, \ldots$ — all divisible by 5.

Step 4. Check whether primes beyond 11 yield simple congruences:

```
q> findcong(partition_gf(500), [13, 17, 19, 23])
[]
```

The empty result confirms that no congruences of the form $p(\ell n + \delta) \equiv 0 \pmod{\ell}$ exist for $\ell \in \{13, 17, 19, 23\}$.

Takeaway. The `findcong` function is a research tool that automates the search for arithmetic congruences in generating function coefficients. Manual verification with `sift` confirms the pattern. This workflow — generate, discover, verify — is a standard research loop in experimental mathematics.

13.3 Jacobi Triple Product Identity

Mathematical context. The Jacobi triple product identity (1829) states

$$\sum_{n=-\infty}^{\infty} z^n q^{n^2} = \prod_{k\geq 1} \left(1 - q^{2k}\right)\left(1 + zq^{2k-1}\right)\left(1 + z^{-1}q^{2k-1}\right)$$

At $z = 1$, this specialises to the theta function

$$\theta_3(q) = \sum_{n=-\infty}^{\infty} q^{n^2} = \prod_{k\geq 1} \left(1 - q^{2k}\right)\left(1 + q^{2k-1}\right)^2$$

which connects the sum-of-squares representation to an infinite product.

Reference: Jacobi, *Fundamenta Nova Theoriae Functionum Ellipticarum* (1829). See also Andrews & Berndt, *Ramanujan's Lost Notebook*, Part I (Springer, 2005).

13.3.1 REPL Workflow

Step 1. Compute $\theta_3(q)$ directly:

```
q> a := theta3(50):
```

Step 2. Build the product side of the identity. The right-hand side at $z = 1$ is $\left(q^2; q^2\right)_\infty \cdot \left(-q; q^2\right)_\infty^2$:

```
q> b := aqprod(1, 1, 2, infinity, 50) * aqprod(-1, 1, 1, infinity, 50)^2:
q> a - b
O(q^50)
```

The difference is zero to 50 terms, confirming the identity.

Step 3. Use `prodmake` to examine the product structure of θ_3:

```
q> prodmake(a, 20)
{1: 0, 2: 1, 3: 0, 4: -1, 5: 0, 6: 1, 7: 0, 8: -1, 9: 0, 10: 1, 11: 0, 12: -1, 13: 0, 14: 1, 15: 0, 16:
-1, 17: 0, 18: 1, 19: 0, 20: -1}
```

The even-index exponents are +1 and the odd-index exponents beyond 1 alternate, consistent with the triple product factorisation.

Step 4. Verify the eta-quotient structure:

```
q> etamake(a, 10)
{1: -2, 2: 5, 4: -2}
```

This says $\theta_3(q) = \frac{\eta(2\tau)^5}{\eta(\tau)^2\eta(4\tau)^2}$, a classical eta-quotient representation.

Takeaway. Multiple verification strategies are available: direct series subtraction, product form analysis via `prodmake`, and eta-quotient identification via `etamake`. Using independent methods strengthens confidence in a conjectured identity.

13.4 Rogers-Ramanujan Identities via Bailey Chains

Mathematical context. The first Rogers-Ramanujan identity states

$$\sum_{n=0}^{\infty} \frac{q^{n^2}}{(q;q)_n} = \prod_{n=1}^{\infty} \frac{1}{(1-q^{5n-4})(1-q^{5n-1})}$$

Rogers (1894) proved this using iterative functional equations, and Ramanujan rediscovered it independently around 1913. In 1947, Bailey introduced the notion of *Bailey pairs* and *Bailey chains*, which provide a systematic framework for generating infinite families of identities from a single seed pair.

Reference: Rogers (1894), Ramanujan (1913); see Andrews, *q-Series: Their Development and Application in Analysis, Number Theory, Combinatorics, Physics and Computer Algebra* (AMS, 1986). For Bailey chains, see Andrews (1984), "Multiple series Rogers-Ramanujan type identities", *Pacific J. Math.* 114, 267–283.

13.4.1 REPL Workflow

Step 1. Start with the Rogers-Ramanujan Bailey pair. The weak lemma produces the (α, β) pair for the base case:

```
q> bailey_weak_lemma(1, 1, 1, 0, 10, 30)
([1, -1, 0, 1, 0, 0, 0, -1, 0, 0], [1, 1, 1, 1, 2, 2, 3, 3, 4, 5])
```

The α sequence shows the sparse signs $(1, -1, 0, 1, 0, ...)$ while β gives the Rogers-Ramanujan coefficients.

Step 2. Apply Bailey's lemma to transform the pair. This lifts the pair to a new pair with modified parameters:

```
q> bailey_apply_lemma(1, 1, 1, 0, 1, 1, 1, 1, 1, 2, 10, 30)
([...], [...])
```

Step 3. Iterate the Bailey chain to depth 2, generating a family of identities from the seed:

```
q> bailey_chain(1, 1, 1, 0, 1, 1, 1, 1, 1, 2, 2, 10, 30)
[[...], [...]]
```

Each element in the chain is a new (α, β) pair that encodes a distinct q-series identity.

Step 4. Use `bailey_discover` to find the proof automatically. Given a target identity as two q-series, `bailey_discover` searches the space of Bailey pairs and chain operations to find a proof path:

```
q> lhs := partition_gf(30):
q> rhs := aqprod(1, 1, 4, infinity, 30) * aqprod(1, 1, 1, infinity, 30):
q> bailey_discover(lhs, rhs, 1, 3, 30)
{found: true, depth: 1, pair: "rogers-ramanujan", ...}
```

Takeaway. Bailey chains generate infinite families of identities from a single seed pair. The `bailey_weak_lemma` / `bailey_apply_lemma` / `bailey_chain` functions let you walk the chain manually, while `bailey_discover` automates the search. This machinery replaces what would be pages of hand computation in classical proofs.

13.5 Hypergeometric Transformations

Mathematical context. The basic hypergeometric series ${}_2\varphi_1$ is

$$ {}^{2}\varphi_1(a, b; c; q, z) = \sum_{n=0}^{\infty} \frac{(a;q)_n (b;q)_n}{(c;q)_n (q;q)_n} z^n $$

Heine (1847) discovered three transformations connecting different ${}_2\varphi_1$ series, analogous to Euler's transformation for the classical Gauss ${}_2F_1$. These transformations are fundamental tools: a series that resists direct summation may become tractable after one or more transformation steps.

Reference: Heine (1847); see Gasper & Rahman, *Basic Hypergeometric Series*, 2nd ed. (Cambridge, 2004), Chapter 1.

13.5.1 REPL Workflow

Step 1. Define a ${}_2\varphi_1$ series with specific parameters. In q-Kangaroo, each parameter $a = \frac{n}{d} \cdot q^p$ is encoded as the triple (n, d, p):

```
q> src := phi([(1,1,2), (1,1,3)], [(1,1,5)], 1, 1, 1, 30):
```

This computes ${}_2\varphi_1(q^2, q^3; q^5; q, q)$ to 30 terms.

Step 2. Apply Heine's first transformation:

```
q> heine1([(1,1,2), (1,1,3)], [(1,1,5)], 1, 1, 1, 30)
(prefactor_series, transformed_series)
```

The result is a pair `(prefactor, transformed)` where the original series equals `prefactor * transformed` and `transformed` is a new ${}_2\varphi_1$ with different parameters.

Step 3. Apply Heine's second transformation to a different series:

```
q> heine2([(1,1,2), (1,1,3)], [(1,1,5)], 1, 1, 1, 30)
(prefactor_series, transformed_series)
```

Step 4. Use BFS search to automatically find a transformation chain between two ${}_2\varphi_1$ series:

```
q> find_transformation_chain(
    [(1,1,2), (1,1,3)], [(1,1,5)], 1, 1, 1,
    [(1,1,2), (1,1,1)], [(1,1,4)], 1, 1, 3,
    3, 30)
{found: true, steps: [{name: "heine1", ...}], depth: 1}
```

The BFS explores the graph of all reachable ${}_2\varphi_1$ series connected by Heine, Sears, and Watson transformations, and returns the shortest path.

Takeaway. Heine's transformations are the q-analogue of Euler's classical hypergeometric transformation. The `find_transformation_chain` function automates what would otherwise be a manual search through transformation identities — a BFS over a graph of equivalent series representations.

13.6 Mock Theta Function Relations

Mathematical context. In his last letter to Hardy (1920), Ramanujan introduced 17 functions he called "mock theta functions" and claimed they shared properties with theta functions but were not themselves theta functions. Zwegers (2002) showed that mock theta functions complete to real-analytic modular forms, placing them in the framework of harmonic Maass forms.

Among the third-order mock theta functions, Watson (1936) proved the relation

$$f(q) + 4\psi(q) = \frac{\theta_4(q)^2}{(q;q)_\infty}$$

connecting the mock theta function $f(q)$ and $\psi(q)$ to classical theta and eta functions.

Reference: Watson (1936), "The Final Problem: An Account of the Mock Theta Functions in the Last Letter of Ramanujan", *J. London Math. Soc.* 11, 55–80. Zwegers (2002), "Mock theta functions", PhD thesis, Utrecht.

13.6.1 REPL Workflow

Step 1. Compute the third-order mock theta functions $f(q)$ and $\psi(q)$:

```
q> mf := mock_theta_f3(50):
q> mpsi := mock_theta_psi3(50):
```

Step 2. Compute the right-hand side: $\theta_4 \frac{(q)^2}{(q;q)_\infty}$:

```
q> t4sq := theta4(50) ^ 2:
q> euler_inv := partition_gf(50):
q> rhs := t4sq * euler_inv:
```

Step 3. Use `findlincombo` to discover the linear relation. We ask: is there a linear combination of $f(q)$ and $\psi(q)$ that equals `rhs`?

```
q> findlincombo(rhs, [mf, mpsi], 0)
[1, 4]
```

The coefficients $[1, 4]$ confirm Watson's relation: $\text{rhs} = 1 \cdot f(q) + 4 \cdot \psi(q)$.

Step 4. Verify independently by direct subtraction:

```
q> rhs - mf - 4 * mpsi
O(q^50)
```

Zero to 50 terms — the relation is confirmed.

Step 5. Explore with the Appell-Lerch sum. Zwegers showed that mock theta functions can be expressed in terms of the Appell-Lerch sum $m(a, z, q)$:

```
q> appell_lerch_m(1, 1, 30)
1 + q + 2*q^2 + 2*q^3 + 4*q^4 + 4*q^5 + 7*q^6 + 8*q^7 + 12*q^8 + 14*q^9 + 20*q^10 + 24*q^11 + 34*q^12 +
40*q^13 + 54*q^14 + 66*q^15 + 86*q^16 + 104*q^17 + 136*q^18 + 164*q^19 + O(q^20)
```

Takeaway. The combination of classical mock theta function computations (`mock_theta_f3`, `mock_theta_psi3`) with relation discovery tools (`findlincombo`) and modern Appell-Lerch machinery (`appell_lerch_m`) provides a computational framework for exploring Ramanujan's mock theta functions. The `findlincombo` function is especially powerful: given candidate series and a target, it discovers the exact coefficients of a linear relation, automating what would be tedious hand calculations.

14 Maple Migration Quick Reference

Users familiar with Frank Garvan's Maple packages (`qseries`, `thetaids`, `ETA`) can use q-Kangaroo as a direct replacement for most operations. The tables below map Maple function names to their q-Kangaroo equivalents. q-Kangaroo also accepts most Maple names as aliases at the REPL prompt, so existing muscle memory transfers immediately.

14.1 Alias Table

The following 17 Maple function names are recognised as aliases in q-Kangaroo. You can type either name at the `q>` prompt:

Maple Name	q-Kangaroo Name	Notes
`numbpart`	`partition_count`	Integer output (not a series)
`rankgf`	`rank_gf`	Dyson rank generating function
`crankgf`	`crank_gf`	Andrews-Garvan crank generating function
`qphihyper`	`phi`	Basic hypergeometric ${}_r\varphi_s$
`qpsihyper`	`psi`	Bilateral hypergeometric ${}_r\psi_s$
`qgauss`	`try_summation`	Tries q-Gauss, q-Vandermonde, q-Chu-Vandermonde
`proveid`	`prove_eta_id`	Eta-quotient identity prover (valence formula)
`qzeil`	`q_zeilberger`	Creative telescoping (shortened alias)
`qzeilberger`	`q_zeilberger`	Creative telescoping (alternative spelling)
`qpetkovsek`	`q_petkovsek`	q-hypergeometric recurrence solver
`qgosper`	`q_gosper`	Indefinite q-hypergeometric summation
`findlincombo_modp`	`findlincombomodp`	Underscore removed
`findhom_modp`	`findhommodp`	Underscore removed
`findhomcombo_modp`	`findhomcombomodp`	Underscore removed
`search_id`	`search_identities`	Expanded name
`g2`	`universal_mock_theta_g2`	Zwegers g_2 short alias
`g3`	`universal_mock_theta_g3`	Zwegers g_3 short alias

All aliases are case-insensitive. You can type `numbpart(100)` or `NUMBPART(100)` at the REPL and both resolve to `partition_count(100)`.

14.2 Complete Function Mapping

The following table maps every Maple function from Garvan's packages to its q-Kangaroo equivalent. Functions marked **Extension** have no Maple counterpart.

14.2.1 Group 1: Pochhammer and q-Binomial

Maple	q-Kangaroo	Notes
`aqprod(a, q, n)`	`aqprod(num, den, pow, n, order)`	Monomial $a = \frac{\text{num}}{\text{den}} q^{\text{pow}}$
`aqprod(a, q, infinity)`	`aqprod(num, den, pow, infinity, order)`	$n =$ `infinity` for $(a;q)_\infty$
`qbin(n, k, q)`	`qbin(n, k, order)`	Gaussian binomial $\binom{n}{k}_q$

14.2.2 Group 2: Named Products

Maple	q-Kangaroo	Notes
`etaq(d, t, q, N)`	`etaq(b, t, order)`	$q^{bt/24}\prod\left(1-q^{bk}\right)^t$
`jacprod(a, b, q, N)`	`jacprod(a, b, order)`	Jacobi triple product $J(a,b)$
`tripleprod(z, b, t, q, N)`	`tripleprod(num, den, pow, order)`	Triple product with z parameter
`quinprod(z, q, N)`	`quinprod(num, den, pow, order)`	Quintuple product
`winquist(a, b, q, N)`	`winquist(a_n, a_d, a_p, b_n, b_d, b_p, order)`	Winquist 10-factor product

14.2.3 Group 3: Theta Functions

Maple	q-Kangaroo	Notes
`theta2(q, N)`	`theta2(order)`	$\theta_2(q)$ in $q^{1/4}$ convention
`theta3(q, N)`	`theta3(order)`	$\theta_3(q) = \sum q^{n^2}$
`theta4(q, N)`	`theta4(order)`	$\theta_4(q) = \sum (-1)^n q^{n^2}$

14.2.4 Group 4: Partition Functions

Maple	q-Kangaroo	Notes
`numbpart(n)`	`partition_count(n)`	Exact $p(n)$ as integer
`seq(numbpart(n), ...)`	`partition_gf(order)`	$1/(q;q)_\infty$ as series
`rankgf(z, q, N)`	`rank_gf(z_num, z_den, order)`	Dyson rank GF $R(z;q)$
`crankgf(z, q, N)`	`crank_gf(z_num, z_den, order)`	Andrews-Garvan crank GF $C(z;q)$
—	`distinct_parts_gf(order)`	**Extension:** $(-q;q)_\infty$
—	`odd_parts_gf(order)`	**Extension:** $1/(q;q^2)_\infty$

Maple	q-Kangaroo	Notes
—	`bounded_parts_gf(max, order)`	**Extension:** parts $\leq$ max

14.2.5 Group 5: Series Analysis

Maple	q-Kangaroo	Notes
`sift(f, m, j)`	`sift(series, m, j)`	Extract a_{mn+j} subsequence
—	`qdegree(series)`	**Extension:** highest q-power
—	`lqdegree(series)`	**Extension:** lowest q-power
`qfactor(f, q, N)`	`qfactor(series)`	Factor into $(1-q^i)$ factors
`prodmake(f, q, N)`	`prodmake(series, n)`	Andrews' product algorithm
`etamake(f, q, N)`	`etamake(series, n)`	Eta-quotient form
—	`jacprodmake(series, n)`	**Extension:** Jacobi product form
`mprodmake(f, q, N)`	`mprodmake(series, n)`	$(1+q^n)$ product form
—	`qetamake(series, n)`	**Extension:** combined eta/q-Pochhammer form

14.2.6 Group 6: Relation Discovery (Exact)

Maple	q-Kangaroo	Notes
`findlincombo(...)`	`findlincombo(target, candidates, topshift)`	target $= \sum c_i f_i$
`findhom(...)`	`findhom(series_list, degree, topshift)`	Homogeneous polynomial relation
`findhomcombo(...)`	`findhomcombo(target, cands, degree, topshift)`	Homogeneous combo
`findnonhom(...)`	`findnonhom(series_list, degree, topshift)`	Non-homogeneous relation
`findnonhomcombo(...)`	`findnonhomcombo(target, cands, deg, topshift)`	Non-homogeneous combo
`findprod(...)`	`findprod(series_list, max_coeff, max_exp)`	Product identity search
`findmaxind(...)`	`findmaxind(series_list, topshift)`	Maximal independent subset
`findpoly(...)`	`findpoly(series_list, degree, topshift)`	Polynomial relation

14.2.7 Group 7: Relation Discovery (Modular)

Maple	q-Kangaroo	Notes
`findcong(f, moduli, q, N)`	`findcong(series, moduli)`	Discover congruences $a_{mn+j} \equiv 0$
`findlincombo_modp(...)`	`findlincombomodp(target, cands, p, topshift)`	Linear relation mod p
`findhom_modp(...)`	`findhommodp(series_list, p, degree, topshift)`	Polynomial mod p
`findhomcombo_modp(...)`	`findhomcombomodp(target, cands, p, deg, topshift)`	Combo mod p

14.2.8 Group 8: Hypergeometric

Maple	q-Kangaroo	Notes
`qphihyper([a1,...], [b1,...], q, z, N)`	`phi(upper, lower, z_n, z_d, z_p, order)`	${}_r\varphi_s$ basic hypergeometric
`qpsihyper([a1,...], [b1,...], q, z, N)`	`psi(upper, lower, z_n, z_d, z_p, order)`	${}_r\psi_s$ bilateral
`qgauss(...)`	`try_summation(upper, lower, z_n, z_d, z_p, order)`	Classical summation formulas
`heine1(...)`	`heine1(upper, lower, z_n, z_d, z_p, order)`	Heine transform I
`heine2(...)`	`heine2(upper, lower, z_n, z_d, z_p, order)`	Heine transform II
`heine3(...)`	`heine3(upper, lower, z_n, z_d, z_p, order)`	Heine transform III
—	`sears_transform(upper, lower, z_n, z_d, z_p, order)`	**Extension:** Sears balanced ${}_4\varphi_3$
—	`watson_transform(upper, lower, z_n, z_d, z_p, order)`	**Extension:** Watson ${}_8\varphi_7 \to {}_4\varphi_3$
—	`find_transformation_chain(...)`	**Extension:** BFS transformation search

14.2.9 Group 9: Identity Proving

Maple	q-Kangaroo	Notes
`proveid(lhs, rhs, level)`	`prove_eta_id(lhs_factors, rhs_factors, level)`	Valence formula proof
—	`search_identities(query, type)`	**Extension:** identity database search
`qgosper(...)`	`q_gosper(upper, lower, z_n, z_d, z_p, q_n, q_d)`	Indefinite summation
`qZeil(...)`	`q_zeilberger(upper, lower, z, n, q, max_order)`	Creative telescoping
—	`verify_wz(upper, lower, z, n, q, max_order, max_k)`	**Extension:** WZ certificate verification
`qPetkovsek(...)`	`q_petkovsek(coefficients, q_num, q_den)`	Recurrence solver
—	`prove_nonterminating(...)`	**Extension:** Chen-Hou-Mu proof

14.2.10 Group 10: Mock Theta, Appell-Lerch, and Bailey

Maple	q-Kangaroo	Notes
—	`mock_theta_f3` … `mock_theta_rho3`	**Extension:** 7 third-order mock theta functions
—	`mock_theta_f0_5` … `mock_theta_cap_f1_5`	**Extension:** 10 fifth-order mock theta functions
—	`mock_theta_cap_f0_7` … `cap_f2_7`	**Extension:** 3 seventh-order mock theta functions
—	`appell_lerch_m(a_pow, z_pow, order)`	**Extension:** Appell-Lerch sum $m(a, z, q)$
`g2(...)`	`universal_mock_theta_g2(a_pow, order)`	Zwegers, $g_2(a; q)$
`g3(...)`	`universal_mock_theta_g3(a_pow, order)`	Zwegers, $g_3(a; q)$
—	`bailey_weak_lemma(...)`	**Extension:** apply Bailey's weak lemma
—	`bailey_apply_lemma(...)`	**Extension:** apply Bailey's full lemma
—	`bailey_chain(...)`	**Extension:** iterate Bailey chain to depth d
—	`bailey_discover(...)`	**Extension:** discover Bailey pair proof

14.3 Key Differences

- **Parameter encoding.** Maple uses symbolic parameters (a, b, z); q-Kangaroo encodes monomials as integer triples `(num, den, pow)` representing $\frac{\text{num}}{\text{den}} \cdot q^{\text{pow}}$. For example, the Maple parameter `q^3` becomes `(1, 1, 3)` and `1/2*q^5` becomes `(1, 2, 5)`.
- **Series display.** q-Kangaroo uses q as the indeterminate (not x). Truncation is shown as $O(q^N)$ where N is the truncation order.
- **Assignment.** Both Maple and q-Kangaroo use `:=` for variable assignment. In q-Kangaroo, `:` at the end of a statement suppresses output (analogous to Maple's terminating colon).
- **No symbolic parameters.** q-Kangaroo works with concrete q-series, not formal symbols. All parameters are evaluated numerically before series construction.
- **No session argument.** In the REPL, functions are called directly (`partition_gf(20)`) without a session object. The Python API requires a `QSession` first argument, but the CLI does not.

15 Index

E

F

G

H

O

P

Q

R

S

T

U

V

W

Z