C++17 Standard Library Quick Reference

A Pocket Guide to Data Structures, Algorithms, and Functions

Second Edition

Peter Van Weert
Marc Gregoire

Apress®

C++17 Standard Library Quick Reference

A Pocket Guide to Data Structures, Algorithms, and Functions

Second Edition

Peter Van Weert
Marc Gregoire

Apress®

C++17 Standard Library Quick Reference: A Pocket Guide to Data Structures, Algorithms, and Functions

Peter Van Weert
Kessel-Lo, Belgium

Marc Gregoire
Meldert, Belgium

ISBN-13 (pbk): 978-1-4842-4922-2
ISBN-13 (electronic): 978-1-4842-4923-9
https://doi.org/10.1007/978-1-4842-4923-9

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail editorial@apress.com; for reprint, paperback, or audio rights, please email bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484249222. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Dedicated to my parents and my brother,
who are always there for me.
Their support and patience helped me
in finishing this book.

—Marc Gregoire

In loving memory of Jeroen.
Your enthusiasm and courage will forever remain
an inspiration to us all.

—Peter Van Weert

Contents

About the Authors

Peter Van Weert is a Belgian software engineer and C++ expert, mainly experienced in large-scale desktop application development. He is passionate about coding, algorithms, and data structures.

Peter received his master of science in computer science summa cum laude with congratulations of the Board of Examiners from the University of Leuven. In 2010, he completed his PhD thesis in Leuven at the research group for declarative languages and artificial intelligence. During his doctoral studies, he was a teaching assistant for courses on software analysis and design, object-oriented programming (Java), and declarative programming (Prolog and Haskell).

After graduating, Peter joined Nikon Metrology to work on industrial metrology software for high-precision 3D laser scanning and point cloud–based inspection. At Nikon, he learned to handle large C++ code bases and gained further proficiency in all aspects of the software development process—skills that serve him well today at Medicim, the software R&D center for dental companies Nobel Biocare, Ormco, and KaVo Kerr. At Medicim, Peter contributes to their next-generation digital platform for dentists, orthodontists, and oral surgeons that offers patient data acquisition from a wide range of hardware, diagnostic functionality, implant planning, and prosthetic design.

In his spare time, Peter writes books on C++ and is a regular speaker at and board member of the Belgian C++ Users Group.

Marc Gregoire is a software architect from Belgium. He graduated from the University of Leuven, Belgium, with a degree in "Burgerlijk ingenieur in de computer wetenschappen" (equivalent to a master of science in engineering in computer science). The year after, he received an advanced master's degree in artificial intelligence, cum laude, at the same university. After his studies, Marc started working for a software consultancy company called Ordina Belgium. As a consultant, he worked for Siemens and Nokia Siemens Networks on critical 2G and 3G software running on Solaris for telecom operators. This required working in international teams stretching from South America and the United States to Europe, the Middle East, Africa, and Asia. Now, Marc is a software architect at Nikon

Metrology (www.nikonmetrology.com), a division of Nikon and a leading provider of precision optical instruments and metrology solutions for 3D geometric inspection.

His main expertise is C/C++, specifically Microsoft VC++ and the MFC framework. He has experience in developing C++ programs running 24/7 on Windows and Linux platforms, for example, KNX/EIB home automation software. In addition to C/C++, Marc also likes C#.

Since April 2007, he has received the annual Microsoft MVP (Most Valuable Professional) award for his Visual C++ expertise.

Marc is the founder of the Belgian C++ Users Group (www.becpp.org), author of *Professional C++* (Wiley), technical editor for numerous books for several publishers, and a member on the CodeGuru forum (as Marc G). He maintains a blog at www.nuonsoft.com/blog/.

About the Technical Reviewer

Christophe Pichaud is a French C/C++ developer based in Paris. Over the course of his career, he has developed large scale server implementations in the banking industry, where he helped build the first French online bank account service (for Banque-Populaire), as well as Retail Services (Société Générale). He's also performed C++ migrations and developed hybrid applications with the .NET stack. Among his past clients are Accenture, Avanade, Sogeti, CapGemini, Palais de Elysée (French Presidency), SNCF, Total, Danone, CACIB, and BNP Paribas. He earned his MCSD.NET certification and currently works for a Microsoft Gold Partner called Devoteam Modern Applications in Paris, a division of Devoteam (www.devoteam.com).

Additionally, he participates in Microsoft Events as speaker for TechDays, and as an MVP at Ask the Expert sessions. He's regularly written C++ technical articles for the French magazine *Programmez* since 2011. He is also the community manager of the ".NET Azure Rangers," which includes 26 members and 9 MVPs and whose activities include speaking, writing and community-building around Microsoft technologies.

When he is not developing software or reading books, Christophe spends his spare time and holidays with his three daughters, Edith, Lisa, and Audrey along with his father Jean-Marc and mother Mireille in the Burgundy region of France.

Introduction

The C++ Standard Library

The C++ Standard Library is a collection of essential classes and functions used by millions of C++ programmers on a daily basis. Being part of the ISO Standard of the C++ Programming Language, an implementation is distributed with virtually every C++ compiler. Code written with the C++ Standard Library is therefore portable across compilers and target platforms.

The Library is more than 25 years old. Its initial versions were heavily inspired by a (then proprietary) C++ library called the *Standard Template Library (STL)*, so much so that many still incorrectly refer to the Standard Library as “the STL.” The STL library pioneered generic programming with templated data structures called *containers* and *algorithms*, glued together with the concept of *iterators*. Most of this work was adapted by the C++ standardization committee, but nevertheless neither library is a true superset of the other.

Today the C++ Standard Library is much more than the containers and algorithms of the STL, though. For decades, it has featured STL-like string classes, extensive localization facilities, and a stream-based I/O library, as well as the entire C Standard Library. Earlier this decade, the C++11 and C++14 editions of the ISO standard have added, among other things, hash map containers, generic smart pointers, a versatile random number generation framework, a powerful regular expression library, more expressive utilities for function-style programming, type traits for template metaprogramming, and a portable concurrency library featuring threads, mutexes, condition variables, and atomic variables. Most recently, C++17 has introduced, among many smaller additions, parallelized algorithms, a file system library, and several key types for day-to-day use (such as `optional<>`, `variant<>`, `any`, and `string_view`). Many of the C++11, C++14, and C++17 additions are based on Boost, a collection of open source C++ libraries.

And this is just the beginning: the C++ community has rarely been as active and alive as in the past decade. The next version of the Standard, tentatively called C++20, is expected to add even more essential classes and functions.

Why This Book?

Needless to say, it is hard to know and remember all the possibilities, details, and intricacies of the vast and growing C++ Standard Library. This handy reference guide offers a condensed, well-structured summary of all essential aspects of the C++ Standard Library and is therefore indispensable to any C++ programmer.

You could consult the Standard itself, but it is written in a very detailed, technical style and is primarily targeted at Library implementors. Moreover, it is very long: the C++ Standard Library chapters alone are nearly 1,000 pages in length, and those on the C Standard Library easily encompass another 200 pages. Other reference guides exist but are often outdated, limited (most cover little more than the STL containers and algorithms), or not much shorter than the Standard itself.

This book covers all important aspects of the C++17 and C18 Standard Libraries, some in more detail than others, and is always driven by their practical usefulness. You will not find page-long, repetitive examples; obscure, rarely used features; or bloated, lengthy explanations that could be summarized in just a few bullets. Instead, this book strives to be exactly that: a summary. Everything you need to know and watch out for in practice is outlined in a compact, to-the-point style, interspersed with practical tips and short, well-chosen, clarifying examples.

Who Should Read This Book?

The book is targeted at all C++ programmers, regardless of their proficiency with the language or the Standard Library. If you are new to C++, its tutorial aspects will quickly bring you up to speed with the C++ Standard Library. Even the most experienced C++ programmer, however, will learn a thing or two from the book and find it an indispensable reference and memory aid. The book does not explain the C++ language or syntax itself, but is accessible to anyone with basic C++ knowledge or programming experience.

What You Will Learn

- How to use the powerful random number generation facilities
- How to work with dates and times
- What smart pointers are and how to use them to prevent memory leaks
- How to use containers to efficiently store and retrieve your data
- How to use algorithms to inspect and manipulate your data
- How lambda expressions allow for elegant use of algorithms
- What functionality the library provides for stream-based I/O
- How to inspect and manipulate files and directories on your file system
- How to work with characters and strings in C++
- How to write localized applications

- How to write safe and efficient multithreaded code using the C++11 concurrency library
- How to correctly handle error conditions and exceptions
- And more!

General Remarks

- All types, classes, functions, and constants of the C++ Standard Library are defined in the `std` namespace (short for *standard*).
- All C++ Standard Library headers must be included using `#include` <*header*> (note: no `.h` suffix!).
- All headers of the C Standard Library are available to C++ programmers in a slightly modified form by including <c*header*> (note the `c` prefix).[1] The most notable difference between the C++ headers and their original C counterparts is that all functionality is defined in the `std` namespace. Whether it is also provided in the global namespace is up to the implementation: portable code should therefore use the `std` namespace at all times.
- This book generally only covers headers of the C Standard Library if the C++ Standard Library does not offer more modern alternatives.
- More advanced, rarely used topics such as custom memory allocators are not covered.

Code Examples

To compile and execute the code examples given throughout the book, all you need is a sufficiently recent C++ compiler. We leave the choice of compiler entirely up to you, and we further assume you can compile and execute basic C++ programs. All examples contain standard, portable C++ code only and should compile with any C++ compiler that is compliant with the C++17 version of the Standard. We occasionally indicate known limitations of major compilers, but this is not a real goal of this book. In case of problems, please consult your implementation's documentation.

Nearly all code examples can be copied as is and put inside the `main()` function of a basic command-line application. Generally, only two headers have to be included to make a code snippet compile: the one being discussed in the context where the

[1]The original C headers may still be included with <*header*`.h`>, but their use is deprecated.

example is given and `<iostream>` for the command-line output statements (explained shortly). If any other header is required, we try to indicate so in the text. Should we forget, the Appendix provides a brief overview of all headers of the Standard Library and their contents. Additionally, you can download compilable source code files for all code snippets from this book from the Apress web site (`www.apress.com/9781484218754`).

Following is the obligatory first example (for once, we show the full program):

```
#include <iostream>

int main() {
   std::cout << "Hello world!" << std::endl;
}
```

Many code samples, including those in earlier chapters, write to the standard output console using `std::cout` and the `<<` *stream insertion operator*, even though these facilities of the C++ I/O library are only discussed in detail in Chapter 5. The stream insertion operator can be used to output virtually all fundamental C++ types, and multiple values can be written on a single line. The so-called *I/O manipulator* `std::endl` outputs the newline character (`\n`) and flushes the output for `std::cout` to the standard console. Straightforward usage of the `std::string` class defined in `<string>` may occur in earlier examples as well. For instance:

```
std::string piString = "PI";
double piValue = 3.14159;
std::cout << piString << " = " << piValue << std::endl;  // PI = 3.14159
```

More details regarding streams and strings are found in Chapters 5 and 6, respectively, but this should suffice to get you through the examples in earlier chapters.

Common Class

The following Person class is used in code examples throughout the book to illustrate the use of user-defined classes together with the Standard Library:

```
#include <string>
#include <ostream>

class Person {
public:
   Person() = default;
   explicit Person(std::string first, std::string last = "",
                   bool isVIP = false)
   : m_first(move(first)), m_last(move(last)), m_isVIP(isVIP) {}

   const std::string& GetFirstName() const { return m_first; }
   void SetFirstName(std::string first) { m_first = move(first); }

   const std::string& GetLastName() const { return m_last; }
   void SetLastName(std::string last) { m_last = move(last); }

   bool IsVIP() const { return m_isVIP; }

private:
   std::string m_first, m_last;
   bool m_isVIP = false;
};

// Comparison operator
bool operator<(const Person& lhs, const Person& rhs) {
   if (lhs.IsVIP() != rhs.IsVIP()) return rhs.IsVIP();
   if (lhs.GetLastName() != rhs.GetLastName())
      return lhs.GetLastName() < rhs.GetLastName();
   return lhs.GetFirstName() < rhs.GetFirstName();
}

// Equality operator
bool operator==(const Person& lhs, const Person& rhs) {
   return lhs.IsVIP() == rhs.IsVIP()
       && lhs.GetFirstName() == rhs.GetFirstName()
       && lhs.GetLastName() == rhs.GetLastName();
}

// Stream insertion operator for output to C++ streams.
// Details of this operator can be found in Chapter 5.
std::ostream& operator<<(std::ostream& os, const Person& person) {
   return os << person.GetFirstName() << ' ' << person.GetLastName();
}
```

CHAPTER 1

Numerics and Math

Common Mathematical Functions <cmath>

The `<cmath>` header defines an extensive collection of common math functions in the `std` namespace. Unless otherwise specified, all functions are overloaded to accept all standard numerical types, with the following rules for determining the return type:

- If all arguments are `float`, the return type is `float` as well. Analogous for `double` and `long double` inputs.
- If mixed types or integers are passed, these numbers are converted to `double`, and a `double` is returned as well. If one of the inputs is a `long double`, `long double` is used instead.

Basic Functions

Function	Description

`abs(x)`
`fabs(x)`
`fabsf(x)`
`fabsl(x)`
	Returns the absolute value of `x`. `abs()` and `fabs()` accept all numeric types; `fabsf()` and `fabsl()` only `float` and `long double`. Starting with C++17, `abs()` no longer converts integers into `double`s (as is conventional for `<cmath>`: see earlier). Instead, it behaves as `abs()` in `<cstdlib>` for integral `x`'s (explained next). C++17

`abs(x)`
`labs(x)`
`llabs(x)`
	Defined by `<cstdlib>`. Returns absolute value for an integral `x`. `abs()` accepts `int`, `long`, or `long long` (smaller integral types are promoted to `int`); `labs()` and `llabs()` only `long` and `long long`. The result has the same (possibly promoted) type as the input.

`fmod(x, y)`
`remainder(x, y)`
	Returns the remainder of x/y. For `fmod()`, the result always has the same sign as `x`; for `remainder()` that is not necessarily true. For example: `mod(1,4)` = `rem(1,4)` = `1`, but `mod(3,4)` = `3` and `rem(3,4)` = `-1`.

`remquo(x, y, *q)`	Returns the same value as `remainder()`. `q` is a pointer to an `int` and receives a value with the sign of x/y and at least the last three bits of the integral quotient itself (rounded to nearest).

(*continued*)

Function	Description

`div(x, y)`
`ldiv(x, y)`
`lldiv(x, y)`
	Defined by `<cstdlib>`. Returns a struct with two members, `quot` and `rem`, containing respectively `x / y` and `x % y` (though often computed with one instruction). `div()` accepts a pair of `int`s, `long`s, or `long long`s; `ldiv()` two `long`s, and `lldiv()` two `long long`s. The results have the same (possibly promoted) type as the inputs.

`fma(x, y, z)`	Computes $(x * y) + z$ in an accurate (better precision and rounding properties than a naïve implementation) and efficient (uses a single hardware instruction if possible) manner.

`fmin(x, y)`
`fmax(x, y)`
	Returns the minimum or maximum of `x` and `y`. `std::min()` and `max()` defined in `<algorithm>` are often more convenient, as they do not convert integers into `double`. These are explained later in this chapter.

`fdim(x, y)`	Returns the positive difference, i.e., $\begin{cases} x - y \; if \; x > y \\ +0 \; if \; x \le y \end{cases}$

`nan(string)`
`nanf(string)`
`nanl(string)`
	Returns a quiet (nonsignaling) NaN (Not-a-Number) of type `double`, `float`, `long double`, respectively, if available (`0` otherwise). The `string` parameter is an implementation-dependent tag that can be used to differentiate between different NaN values. Both `""` and `nullptr` are valid and result in a generic quiet NaN.

Exponential and Logarithmic Functions

Function	Formula	Function	Formula	Function	Formula
`exp(x)`	e^x	`exp2(x)`	2^x	`expm1(x)`	$e^x - 1$
`log(x)`	$\ln x = \log_e x$	`log10(x)`	$\log_{10} x$	`log2(x)`	$\log_2 x$
`log1p(x)`	$\ln(1 + x)$				

Power Functions

Function	Formula	Function	Formula
`pow(x, y)`	x^y	`sqrt(x)`	$\sqrt{x}$
`hypot(x, y)`	$\sqrt{x^2 + y^2}$	`cbrt(x)`	$\sqrt[3]{x}$
`hypot(x, y, z)`	$\sqrt{x^2 + y^2 + z^2}$ C++17		

Trigonometric and Hyperbolic Functions

`<cmath>` provides all basic trigonometric (`sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()`) and hyperbolic functions (`sinh()`, `cosh()`, `tanh()`, `asinh()`, `acosh()`, `atanh()`). All angles are expressed in radians.

The lesser-known trigonometric function `atan2()` is available as well. You use it to compute the angle between a vector (`x`, `y`) and the positive X axis. `atan2(y, x)` is similar to `atan(y / x)` except that its result correctly reflects the quadrant the vector is in (and that it also works if `x` is zero). Essentially, by dividing `y` by `x` in `atan(y / x)`, one loses information regarding the sign of `x` and `y`.

Integral Rounding of Floating-Point Numbers

Function	Description

`ceil(x)`
`floor(x)`
	Rounds up/down to an integer. That is, returns the nearest integer that is not less/not greater than `x`.

`trunc(x)`	Returns the nearest integer not greater in absolute value than `x`.

`round(x)`
`lround(x)`
`llround(x)`
	Returns the integral value nearest to `x`, rounding halfway cases away from zero. The return type of `round()` is based as usual on the type of `x`, while `lround()` returns `long`, and `llround()` returns `long long`.

`nearbyint(x)`	Returns the integral value nearest to `x` as a floating-point type. The current rounding mode is used: see `round_style` in the section on arithmetic type properties later in this chapter.

`rint(x)`
`lrint(x)`
`llrint(x)`
	Returns the integral value nearest to `x`, using the current rounding mode. The return type of `rint()` is based as usual on the type of `x`, while `lrint()` returns `long`, and `llrint()` returns `long long`.

Floating-Point Manipulation Functions

Function	Description
`modf(x, *p)`	Breaks the value of `x` into an integral and fractional part. The latter is returned, the former is stored in `p`, both with the same sign as `x`. The return type is based on that of `x` as usual, and `p` must point to a value of the same type as this return type.
`frexp(x, *exp)`	Breaks the value of `x` into a normalized fraction with an absolute value in the range [0.5, 1) or equal to zero (the return value), and an integral power of 2 (stored in `exp`), with $x = fraction * 2^{exp}$.
`logb(x)`	Returns the floating-point exponent of `x`, i.e., $\log_{radix}\|x\|$, with *radix* the base used to represent floating-point values (2 for all standard numerical types, hence the name 'binary logarithm').
`ilogb(x)`	Same as `logb(x)` but the result is truncated to a signed `int`.

(*continued*)

Function	Description
`ldexp(x, n)`	Returns $x * 2^n$ (with `n` an `int`).

`scalbn(x, n)`
`scalbln(x, n)`
	Returns $x * radix^n$ (with `n` an `int` for `scalbn()` and a `long` for `scalbln()`). Radix is the base used to represent floating-point values (2 for all standard C++ numerical types).

`nextafter(x, y)`
`nexttoward(x, y)`
	Returns the next representable value after `x` in the direction of `y`. Returns `y` if `x` equals `y`. For `nexttoward()`, the type of `y` is always `long double`.

`copysign(x, y)`	Returns a value with the absolute value of `x` and the sign of `y`.

Classification and Comparison Functions

Function	Description
`fpclassify(x)`	Classifies the floating-point value `x`: returns an `int` equal to `FP_INFINITE`, `FP_NAN`, `FP_NORMAL`, `FP_SUBNORMAL`, `FP_ZERO`, or an implementation-specific category.
`isfinite(x)`	Returns `true` if `x` is finite, i.e., normal, subnormal (denormalized), or zero, but not infinite or not-a-number.
`isinf(x)`	Returns `true` if `x` is positive or negative infinity.
`isnan(x)`	Returns `true` if `x` is not-a-number.
`isnormal(x)`	Returns `true` if `x` is normal, i.e., neither zero, subnormal (denormalized), infinite, nor not-a-number.
`signbit(x)`	Returns a nonzero value if `x` is negative.

`isgreater(x, y)`
`isgreaterequal(x, y)`
`isless(x, y)`
`islessequal(x, y)`
`islessgreater(x, y)`
	Compares `x` and `y`. The names are self-explanatory, except `islessgreater()` which returns `true` if $x < y \|\| x > y$. Note that this is not the same as `!=`, as, e.g., `nan("") != nan("")` is `true`, but not `islessgreater(nan(""), nan(""))`.

`isunordered(x, y)`	Returns whether `x` and `y` are unordered, i.e., whether one or both are not-a-number.

gcd/lcm C++17 <numeric>

The functions `gcd()` and `lcm()` compute the greatest common divisor and least common multiple, respectively. They are defined as follows:

```
template<typename M, typename N>
constexpr std::common_type_t<M, N> gcd(M, N);
```

```
template<typename M, typename N>
constexpr std::common_type_t<M, N> lcm(M, N);
```

Both `M` and `N` must be integer types. As explained in Chapter 2, `std::common_type_t<M, N>` is a so-called *type trait*, which in this case results in a type that both `M` and `N` can implicitly be converted to. Concretely, the common type of two integer types `M` and `N` is determined by the following rules (applied in order):

- If `N` and `M` are equal, their common type is that same type as well.
- If `N` and `M` are both smaller than `int`, their common type is `int`.
- If the size of `N` and `M` differs, their common type is the largest type.
- Otherwise, the common type is the one that is `unsigned`.

Error Handling

The mathematical functions from `<cmath>` can report errors in two ways depending on the value of `math_errhandling` (defined in `<cmath>`, although not in the `std` namespace). It has an integral type and can have one of the following values or their bitwise OR combination:

- `MATH_ERRNO`: Use the global `errno` variable (see Chapter 8).
- `MATH_ERREXCEPT`: Use the floating-point environment, `<cfenv>`, not further discussed in this book.

Special Mathematical Functions C++17 `<cmath>`

C++17 adds a collection of specialized mathematical functions. All of these are available in multiple overloads. In the following table, the functions without an asterisk always return a `double`. For the functions marked with an asterisk, the return type is always `double`, unless one of its arguments is a `long double`, then the return type is `long double` as well.

Additionally, there are two extra versions of each function with a postfix `f` or `l`. These additional functions accept `float`s and return a `float` (`f` postfix), or accept `long double`s and return a `long double` (`l` postfix). For example, `assoc_laguerre()`, `assoc_laguerref()`, and `assoc_laguerrel()`.

Explaining all the details of these mathematical functions falls outside the scope of this book. The following table just shows the mathematical formula for each function. Please consult a mathematical reference for more details.

■ **Note** At the time of writing, libc++, the implementation that ships with the Clang compiler, has not implemented these special mathematical functions yet.

Bessel Functions

Function	Description
`cyl_bessel_j(`v`, x)*`	Computesv thev cylindrical Bessel function of the first kind: $J_v(x)=\sum_{k=0}^{\infty}\frac{(-1)^k(x/2)^{v+2k}}{k!\Gamma(v+k+1)},\ \text{for } \lvert x\rvert \geq 0$
`cyl_neumann(`v`, x)*`	Computes the cylindrical Neumann function, also known as the cylindrical Bessel function of the second kind: $N_v(x)=\begin{cases}\frac{J_v(x)\cos v\pi - J_{-v}(x)}{\sin v\pi}, & \text{for } x\geq 0 \text{ and non-integral } v\\ \lim_{\mu\to v}\left(\frac{J_\mu(x)\cos\mu\pi - J_{-\mu}(x)}{\sin\mu\pi}\right), & \text{for } x\geq 0 \text{ and integral } v\end{cases}$
`cyl_bessel_i(`v`, x)*`	Computes the regular modified cylindrical Bessel function: $I_v(x)=i^{-v}J_v(ix)=\sum_{k=0}^{\infty}\frac{(x/2)^{v+2k}}{k!\Gamma(v+k+1)},\ \text{for } \lvert x\rvert\geq 0$
`cyl_bessel_k(`v`, x)*`	Computes the irregular modified cylindrical Bessel function: $K_v(x)=(\pi/2)i^{v+1}\left(J_v(ix)+iN_v(ix)\right)$ $=\begin{cases}\frac{\pi}{2}\frac{I_{-v}(x)-I_v(x)}{\sin v\pi}, & \text{for } x\geq 0 \text{ and non-integral } v\\ \frac{\pi}{2}\lim_{\mu\to v}\left(\frac{I_{-\mu}(x)-I_\mu(x)}{\sin\mu\pi}\right), & \text{for } x\geq 0 \text{ and integral } v\end{cases}$
`sph_bessel(n, x)`	Computes the spherical Bessel function of the first kind: $j_n(x)=\left(\frac{\pi}{2x}\right)^{1/2}J_{n+1/2}(x),\ \text{for } x\geq 0$
`sph_neumann(n, x)`	Computes the spherical Neumann function, also known as the spherical Bessel function of the second kind: $n_x(x)=\left(\frac{\pi}{2x}\right)^{1/2}N_{n+1/2}(x),\ \text{for } x\geq 0$

Polynomials

Function	Description
`legendre(l, x)`	Computes the Legendre polynomial of the first kind: $P_l(x) = \frac{1}{2^l l!}\frac{d^l}{dx^l}(x^2-1)^l, \text{ for } \lvert x\rvert \le 1$
`assoc_legendre(l, m, x)`	Computes the associated Legendre function: $P_l^m(x) = (1-x^2)^{m/2}\frac{d^m}{dx^m}P_l(x), \text{ for } \lvert x\rvert \le 1$
`sph_legendre(l, m, θ)`	Computes the spherical associated Legendre function $Y_l^m(\theta,0)$, where: $Y_l^m(\theta,\phi) = (-1)^m\left(\frac{(2l+1)}{4\pi}\frac{(l-m)!}{(l+m)!}\right)^{1/2} P_l^m(\cos\theta)e^{im\phi},$ for $\lvert m\rvert \le l$
`laguerre(n, x)`	Computes the Laguerre polynomial of order *n* at point *x*: $L_n(x) = \frac{e^x}{n!}\frac{d^n}{dx^n}(x^n e^{-x}), \text{ for } x \ge 0$
`assoc_laguerre(n, m, x)`	Computes the associated Laguerre polynomial: $L_n^m(x) = (-1)^m\frac{d^m}{dx^m}L_{n+m}(x), \text{ for } x \ge 0$
`hermite(n, x)`	Computes the Hermite polynomial of order *n* at point *x*: $H_n(x) = (-1)^n e^{x^2}\frac{d^n}{dx^n}e^{-x^2}$

Elliptic Integrals

Function	Description
`ellint_1(k, ϕ)*`	Computes the incomplete elliptic integral of the first kind: $F(k,\phi) = \int_0^\phi \frac{d\theta}{\sqrt{1-k^2\sin^2\theta}}, \text{ for } \lvert k\rvert \le 1$
`comp_ellint_1(k)`	Computes the complete elliptic integral of the first kind: $F\left(k,\frac{\pi}{2}\right), \text{ for } \lvert k\rvert \le 1$
`ellint_2(k, ϕ)*`	Computes the incomplete elliptic integral of the second kind: $E(k,\phi) = \int_0^\phi \sqrt{1-k^2\sin^2\theta}\,d\theta, \text{ for } \lvert k\rvert \le 1$

(*continued*)

Function	Description
`comp_ellint_2(k)`	Computes the complete elliptic integral of the second kind: $E\left(k, \frac{\pi}{2}\right)$, for $\lvert k \rvert \le 1$
`ellint_3(k, ν, ϕ)*`	Computes the incomplete elliptic integral of the third kind: $\sqcap(\nu, k, \phi) = \int_0^{\phi} \frac{d\theta}{\left(1 - \nu \sin^2\theta\right)\sqrt{1 - k^2 \sin^2\theta}}$, for $\lvert k \rvert \le 1$
`comp_ellint_3(k, ν)*`	Computes the complete elliptic integral of the third kind: $\sqcap\left(k, \nu, \frac{\pi}{2}\right)$, for $\lvert k \rvert \le 1$

Exponential Integrals

Function	Description
`expint(x)`	Computes the exponential integral: $Ei(x) = -\int_{-x}^{\infty} \frac{e^{-t}}{t} dt$

Error Functions

Function	Description
`erf(x)`	Computes the error function of a given value: $erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$
`erfc(x)`	Computes the complement of the error function of a given value: $erfc(x) = 1 - erf(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt$

Gamma Functions

Function	Description
`tgamma(x)`	Computes the "true gamma" of a given value: $\Gamma(x) = \int_0^{\infty} t^{x-1} e^{-t} dt$
`lgamma(x)`	Computes: $\ln(\lvert \Gamma(x) \rvert)$