Stan Functions Reference

Version 2.35

Stan Development Team

Table of Contents

Overview

This is the reference for the functions defined in the Stan math library and available in the Stan programming language.

For more information the Stan language and inference engines and how to use Stan for Bayesian inference, see

- the **Stan User's Guide**. The Stan user's guide provides example models and programming techniques for coding statistical models in Stan. It also serves as an example-driven introduction to Bayesian modeling and inference:
- the **Stan Reference Manual**. Stan's modeling language is shared across all of its interfaces. The Stan Language Reference Manual provides a concise definition of the language syntax for all elements in the language together with an overview of the inference algorithms and posterior inference tools.

Part I

Built-in Functions

1. Void Functions

Stan supports a few special statements for printing or for signaling an issue with the program.

Although `print`, `reject`, and `fatal_error` appear to have the syntax of functions, they are actually special kinds of statements with slightly different form and behavior than other functions. First, they are the constructs that allow a variable number of arguments. Second, they are the the only constructs to accept string literals (e.g., `"hello world"`) as arguments. Third, they have no effect on the log density function and operate solely through side effects.

The special keyword `void` is used for their return type because they behave like variadic functions with void return type, even though they are special kinds of statements built in to the language.

1.1. Print statement

Printing has no effect on the model's log probability function. Its sole purpose is the side effect (i.e., an effect not represented in a return value) of arguments being printed to whatever the standard output stream is connected to (e.g., the terminal in command-line Stan or the R console in RStan).

`void` **`print`**`(T1 x1,..., TN xN)`
Print the values denoted by the arguments `x1` through `xN` on the output message stream. There are no spaces between items in the print, but a line feed (LF; Unicode U+000A; C++ literal `'\n'`) is inserted at the end of the printed line. The types `T1` through `TN` can be any of Stan's built-in numerical types or double quoted strings of characters (bytes).

Available since 2.1

1.2. Reject statement

The reject statement has the same syntax as the print statement, accepting an arbitrary number of arguments of any type (including string literals). The effect of executing a reject statement is to throw an exception internally that terminates the current iteration with a rejection (the behavior of which will depend on the algorithmic context in which it occurs).

`void` **`reject`**`(T1 x1,..., TN xN)`
Reject the current iteration and print the values denoted by the arguments `x1` through `xN` on the output message stream. There are no spaces between items in the print, but a line feed (LF; Unicode U+000A; C++ literal `'\n'`) is inserted at the end of the printed line. The types `T1` through `TN` can be any of Stan's built-in numerical types or double quoted strings of characters (bytes).

Available since 2.18

1.3. Fatal error statement

The fatal error statement has the same syntax as the print and reject statements, accepting an arbitrary number of arguments of any type (including string literals). The effect of executing a fatal_error statement is to throw an exception internally that terminates the algorithm completely. It can be viewed as an unrecoverable version of reject, and as such should be used only when exiting the algorithm is the only option.

`void` **`fatal_error`**`(T1 x1,..., TN xN)`
Print the values denoted by the arguments `x1` through `xN` on the output message stream and then exit the currently running algorithm. There are no spaces between items in the print, but a line feed (LF; Unicode U+000A; C++ literal `'\n'`) is inserted at the end of the printed line. The types `T1` through `TN` can be any of Stan's built-in numerical types or double quoted strings of characters (bytes).

Available since 2.35

2. Integer-Valued Basic Functions

This chapter describes Stan's built-in function that take various types of arguments and return integer values.

2.1. Integer-valued arithmetic operators

Stan's arithmetic is based on standard double-precision C++ integer and floating-point arithmetic. If the arguments to an arithmetic operator are both integers, as in `2 + 2`, integer arithmetic is used. If one argument is an integer and the other a floating-point value, as in `2.0 + 2` and `2 + 2.0`, then the integer is promoted to a floating point value and floating-point arithmetic is used.

Integer arithmetic behaves slightly differently than floating point arithmetic. The first difference is how overflow is treated. If the sum or product of two integers overflows the maximum integer representable, the result is an undesirable wraparound behavior at the bit level. If the integers were first promoted to real numbers, they would not overflow a floating-point representation. There are no extra checks in Stan to flag overflows, so it is up to the user to make sure it does not occur.

Secondly, because the set of integers is not closed under division and there is no special infinite value for integers, integer division implicitly rounds the result. If both arguments are positive, the result is rounded down. For example, `1 / 2` evaluates to 0 and `5 / 3` evaluates to 1.

If one of the integer arguments to division is negative, the latest C++ specification (C++11), requires rounding toward zero. This would have `1 / 2` and `-1 / 2` evaluate to 0, `-7 / 2` evaluate to -3, and `7 / 2` evaluate to 3. Before the C++11 specification, the behavior was platform dependent, allowing rounding up or down. All compilers recent enough to be able to deal with Stan's templating should follow the C++11 specification, but it may be worth testing if you are not sure and plan to use integer division with negative values.

Unlike floating point division, where `1.0 / 0.0` produces the special positive infinite value, integer division by zero, as in `1 / 0`, has undefined behavior in the C++ standard. For example, the clang++ compiler on Mac OS X returns 3764, whereas the g++ compiler throws an exception and aborts the program with a warning. As with overflow, it is up to the user to make sure integer divide-by-zero does not occur.

Binary infix operators

Operators are described using the C++ syntax. For instance, the binary operator of addition, written `X + Y`, would have the Stan signature `int operator+(int, int)` indicating it takes two real arguments and returns a real value. As noted previously, the value of integer division is platform-dependent when rounding is platform dependent before C++11; the descriptions below provide the C++11 definition.

`int` **`operator+`**`(int x, int y)`
The sum of the addends x and y

$$\text{operator+}(x, y) = (x + y)$$

Available since 2.0

`int` **`operator-`**`(int x, int y)`
The difference between the minuend x and subtrahend y

$$\text{operator-}(x, y) = (x - y)$$

Available since 2.0

`int` **`operator*`**`(int x, int y)`
The product of the factors x and y

$$\text{operator*}(x, y) = (x \times y)$$

Available since 2.0

`int` **`operator/`**`(int x, int y)`
The integer quotient of the dividend x and divisor y

$$\text{operator}/(x, y) = \begin{cases} \lfloor x/y \rfloor & \text{if } x/y \geq 0 \\ -\lfloor \text{floor}(-x/y) \rfloor & \text{if } x/y < 0. \end{cases}$$

deprecated; - use `operator%/%` instead.

Available since 2.0, deprecated in 2.24

`int` **`operator%/%`**`(int x, int y)`
The integer quotient of the dividend x and divisor y

$$\text{operator\%/\%}(x, y) = \begin{cases} \lfloor x/y \rfloor & \text{if } x/y \geq 0 \\ -\lfloor \text{floor}(-x/y) \rfloor & \text{if } x/y < 0. \end{cases}$$

Available since 2.24

`int` **`operator%`**`(int x, int y)`
x modulo y, which is the positive remainder after dividing x by y. If both x and y are non-negative, so is the result; otherwise, the sign of the result is platform dependent.

$$\text{operator\%}(x, y) \;=\; x \bmod y \;=\; x - y * \lfloor x/y \rfloor$$

Available since 2.13

Unary prefix operators

`int` **`operator-`**`(int x)`
The negation of the subtrahend x

$$\text{operator-}(x) = -x$$

Available since 2.0

`T` **`operator-`**`(T x)`
Vectorized version of `operator-`. If `T x` is a (possibly nested) array of integers, `-x` is the same shape array where each individual integer is negated.

Available since 2.31

`int` **`operator+`**`(int x)`
This is a no-op.

$$\text{operator+}(x) = x$$

Available since 2.0

2.2. Absolute functions

`T` **`abs`**`(T x)`
The absolute value of x.

This function works elementwise over containers such as vectors. Given a type `T` which is `int`, or an array of `int`s, `abs` returns the same type where each element has had its absolute value taken.

Available since 2.0, vectorized in 2.30

`int` **`int_step`**`(int x)`

`int` **`int_step`**`(real x)`
Return the step function of x as an integer,

$$\text{int\_step}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \text{ or } x \text{ is } NaN \end{cases}$$

Warning: `int_step(0)` and `int_step(NaN)` return 0 whereas `step(0)` and `step(NaN)` return 1.

See the warning in section step functions about the dangers of step functions applied to anything other than data.

Available since 2.0

2.3. Bound functions

`int` **`min`**`(int x, int y)`
Return the minimum of x and y.

$$\min(x, y) = \begin{cases} x & \text{if } x < y \\ y & \text{otherwise} \end{cases}$$

Available since 2.0

`int` **`max`**`(int x, int y)`
Return the maximum of x and y.

$$\max(x, y) = \begin{cases} x & \text{if } x > y \\ y & \text{otherwise} \end{cases}$$

Available since 2.0

2.4. Size functions

`int` **`size`**`(int x)`

`int` **`size`**`(real x)`

Return the size of x which for scalar-valued x is 1

Available since 2.26

2.5. Casting functions

It is possible to cast real numbers to integers as long as the real value is `data`. See data only qualifiers in the Stan Reference Manual.

`int` **`to_int`**`(data real x)`

Return the value `x` truncated to an integer. This will throw an error if the value of `x` is too big to represent as a 32-bit signed integer.

This is similar to `trunc` (see Rounding functions) but the return type is of type `int`. For example, `to_int(3.9)` is `3`, and `to_int(-3.9)` is `-3`.

Available since 2.31

`I` **`to_int`**`(data T x)`

The vectorized version of `to_int`. This function accepts a (possibly nested) array of reals and returns an array of the same shape where each element has been truncated to an integer.

Available since 2.31

3. Real-Valued Basic Functions

This chapter describes built-in functions that take zero or more real or integer arguments and return real values.

3.1. Vectorization of real-valued functions

Although listed in this chapter, many of Stan's built-in functions are vectorized so that they may be applied to any argument type. The vectorized form of these functions is not any faster than writing an explicit loop that iterates over the elements applying the function—it's just easier to read and write and less error prone.

Unary function vectorization

Many of Stan's unary functions can be applied to any argument type. For example, the exponential function, `exp`, can be applied to `real` arguments or arrays of `real` arguments. Other than for integer arguments, the result type is the same as the argument type, including dimensionality and size. Integer arguments are first promoted to real values, but the result will still have the same dimensionality and size as the argument.

Real and real array arguments

When applied to a simple real value, the result is a real value. When applied to arrays, vectorized functions like `exp()` are defined elementwise. For example,

```
// declare some variables for arguments
real x0;
array[5] real x1;
array[4, 7] real x2;
// ...
// declare some variables for results
real y0;
array[5] real y1;
array[4, 7] real y2;
// ...
// calculate and assign results
y0 = exp(x0);
y1 = exp(x1);
y2 = exp(x2);
```

When exp is applied to an array, it applies elementwise. For example, the statement above,

```
y2 = exp(x2);
```

produces the same result for y2 as the explicit loop

```
for (i in 1:4) {
  for (j in 1:7) {
    y2[i, j] = exp(x2[i, j]);
  }
}
```

Vector and matrix arguments

Vectorized functions also apply elementwise to vectors and matrices. For example,

```
vector[5] xv;
row_vector[7] xrv;
matrix[10, 20] xm;

vector[5] yv;
row_vector[7] yrv;
matrix[10, 20] ym;

yv = exp(xv);
yrv = exp(xrv);
ym = exp(xm);
```

Arrays of vectors and matrices work the same way. For example,

```
array[12] matrix[17, 93] u;

array[12] matrix[17, 93] z;

z = exp(u);
```

After this has been executed, z[i, j, k] will be equal to exp(u[i, j, k]).

Integer and integer array arguments

Integer arguments are promoted to real values in vectorized unary functions. Thus if n is of type int, exp(n) is of type real. Arrays work the same way, so that if n2 is a one dimensional array of integers, then exp(n2) will be a one-dimensional array of reals with the same number of elements as n2. For example,

```
array[23] int n1;
array[23] real z1;
z1 = exp(n1);
```

It would be illegal to try to assign `exp(n1)` to an array of integers; the return type is a real array.

Binary function vectorization

Like the unary functions, many of Stan's binary functions have been vectorized, and can be applied elementwise to combinations of both scalars or container types.

Scalar and scalar array arguments

When applied to two scalar values, the result is a scalar value. When applied to two arrays, or combination of a scalar value and an array, vectorized functions like `pow()` are defined elementwise. For example,

```
// declare some variables for arguments
real x00;
real x01;
array[5] real x10;
array[5]real x11;
array[4, 7] real x20;
array[4, 7] real x21;
// ...
// declare some variables for results
real y0;
array[5] real y1;
array[4, 7] real y2;
// ...
// calculate and assign results
y0 = pow(x00, x01);
y1 = pow(x10, x11);
y2 = pow(x20, x21);
```

When `pow` is applied to two arrays, it applies elementwise. For example, the statement above,

```
y2 = pow(x20, x21);
```

produces the same result for `y2` as the explicit loop

```
for (i in 1:4) {
  for (j in 1:7) {
    y2[i, j] = pow(x20[i, j], x21[i, j]);
  }
}
```

Alternatively, if a combination of an array and a scalar are provided, the scalar value is broadcast to be applied to each value of the array. For example, the following statement:

```
y2 = pow(x20, x00);
```

produces the same result for `y2` as the explicit loop:

```
for (i in 1:4) {
  for (j in 1:7) {
    y2[i, j] = pow(x20[i, j], x00);
  }
}
```

Vector and matrix arguments

Vectorized binary functions also apply elementwise to vectors and matrices, and to combinations of these with scalar values. For example,

```
real x00;
vector[5] xv00;
vector[5] xv01;
row_vector[7] xrv;
matrix[10, 20] xm;

vector[5] yv;
row_vector[7] yrv;
matrix[10, 20] ym;

yv = pow(xv00, xv01);
yrv = pow(xrv, x00);
ym = pow(x00, xm);
```

Arrays of vectors and matrices work the same way. For example,

```
array[12] matrix[17, 93] u;
```

```
array[12] matrix[17, 93] z;

z = pow(u, x00);
```

After this has been executed, `z[i, j, k]` will be equal to `pow(u[i, j, k], x00)`.

Input & return types

Vectorised binary functions require that both inputs, unless one is a real, be containers of the same type and size. For example, the following statements are legal:

```
vector[5] xv;
row_vector[7] xrv;
matrix[10, 20] xm;

vector[5] yv = pow(xv, xv)
row_vector[7] yrv = pow(xrv, xrv)
matrix[10, 20] = pow(xm, xm)
```

But the following statements are not:

```
vector[5] xv;
vector[7] xv2;
row_vector[5] xrv;

// Cannot mix different types
vector[5] yv = pow(xv, xrv)

// Cannot mix different sizes of the same type
vector[5] yv = pow(xv, xv2)
```

While the vectorized binary functions generally require the same input types, the only exception to this is for binary functions that require one input to be an integer and the other to be a real (e.g., `bessel_first_kind`). For these functions, one argument can be a container of any type while the other can be an integer array, as long as the dimensions of both are the same. For example, the following statements are legal:

```
vector[5] xv;
matrix[5, 5] xm;
array[5] int xi;
array[5, 5] int xii;
```

```
vector[5] yv = bessel_first_kind(xi, xv);
matrix[5, 5] ym = bessel_first_kind(xii, xm);
```

Whereas these are not:

```
vector[5] xv;
matrix[5, 5] xm;
array[7] int xi;

// Dimensions of containers do not match
vector[5] yv = bessel_first_kind(xi, xv);

// Function requires first argument be an integer type
matrix[5, 5] ym = bessel_first_kind(xm, xm);
```

3.2. Mathematical constants

Constants are represented as functions with no arguments and must be called as such. For instance, the mathematical constant π must be written in a Stan program as `pi()`.

`real` **`pi`**`()`
π, the ratio of a circle's circumference to its diameter

Available since 2.0

`real` **`e`**`()`
e, the base of the natural logarithm

Available since 2.0

`real` **`sqrt2`**`()`
The square root of 2

Available since 2.0

`real` **`log2`**`()`
The natural logarithm of 2

Available since 2.0

`real` **`log10`**`()`
The natural logarithm of 10

Available since 2.0

3.3. Special values

`real` **`not_a_number`**`()`
Not-a-number, a special non-finite real value returned to signal an error

Available since 2.0

`real` **`positive_infinity`**`()`
Positive infinity, a special non-finite real value larger than all finite numbers

Available since 2.0

`real` **`negative_infinity`**`()`
Negative infinity, a special non-finite real value smaller than all finite numbers

Available since 2.0

`real` **`machine_precision`**`()`
The smallest number x such that $(x + 1) \neq 1$ in floating-point arithmetic on the current hardware platform

Available since 2.0

3.4. Log probability function

The basic purpose of a Stan program is to compute a log probability function and its derivatives. The log probability function in a Stan model outputs the log density on the unconstrained scale. A log probability accumulator starts at zero and is then incremented in various ways by a Stan program. The variables are first transformed from unconstrained to constrained, and the log Jacobian determinant added to the log probability accumulator. Then the model block is executed on the constrained parameters, with each sampling statement (~) and log probability increment statement (`increment_log_prob`) adding to the accumulator. At the end of the model block execution, the value of the log probability accumulator is the log probability value returned by the Stan program.

Stan provides a special built-in function `target()` that takes no arguments and returns the current value of the log probability accumulator. This function is primarily useful for debugging purposes, where for instance, it may be used with a print statement to display the log probability accumulator at various stages of execution to see where it becomes ill defined.

`real` **`target`**`()`
Return the current value of the log probability accumulator.

Available since 2.10

`target` acts like a function ending in `_lp`, meaning that it may only may only be used in the model block.

3.5. Logical functions

Like C++, BUGS, and R, Stan uses 0 to encode false, and 1 to encode true. Stan supports the usual boolean comparison operations and boolean operators. These all have the same syntax and precedence as in C++; for the full list of operators and precedences, see the reference manual.

Comparison operators

All comparison operators return boolean values, either 0 or 1. Each operator has two signatures, one for integer comparisons and one for floating-point comparisons. Comparing an integer and real value is carried out by first promoting the integer value.

`int` **`operator<`**`(int x, int y)`

`int` **`operator<`**`(real x, real y)`
Return 1 if x is less than y and 0 otherwise.

$$\text{operator<}(x, y) = \begin{cases} 1 & \text{if } x < y \\ 0 & \text{otherwise} \end{cases}$$

Available since 2.0

`int` **`operator<=`**`(int x, int y)`

`int` **`operator<=`**`(real x, real y)`
Return 1 if x is less than or equal y and 0 otherwise.

$$\text{operator<=}(x, y) = \begin{cases} 1 & \text{if } x \leq y \\ 0 & \text{otherwise} \end{cases}$$

Available since 2.0

`int` **`operator>`**`(int x, int y)`

`int` **`operator>`**`(real x, real y)`

Return 1 if x is greater than y and 0 otherwise.

$$\text{operator>}(x, y) = \begin{cases} 1 & \text{if } x > y \\ 0 & \text{otherwise} \end{cases}$$

Available since 2.0

`int` **`operator>=`**`(int x, int y)`

`int` **`operator>=`**`(real x, real y)`
Return 1 if x is greater than or equal to y and 0 otherwise.

$$\text{operator>=}(x, y) = \begin{cases} 1 & \text{if } x \geq y \\ 0 & \text{otherwise} \end{cases}$$

Available since 2.0

`int` **`operator==`**`(int x, int y)`

`int` **`operator==`**`(real x, real y)`
Return 1 if x is equal to y and 0 otherwise.

$$\text{operator==}(x, y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

Available since 2.0

`int` **`operator!=`**`(int x, int y)`

`int` **`operator!=`**`(real x, real y)`
Return 1 if x is not equal to y and 0 otherwise.

$$\text{operator!=}(x, y) = \begin{cases} 1 & \text{if } x \neq y \\ 0 & \text{otherwise} \end{cases}$$

Available since 2.0

Boolean operators

Boolean operators return either 0 for false or 1 for true. Inputs may be any real or integer values, with non-zero values being treated as true and zero values treated as false. These operators have the usual precedences, with negation (not) binding the most tightly, conjunction the next and disjunction the weakest; all of the operators bind more tightly than the comparisons. Thus an expression such as `!a && b` is interpreted as `(!a) && b`, and `a < b || c >= d && e != f` as `(a < b) || (((c >= d) && (e != f)))`.

`int` **`operator!`**`(int x)`
Return 1 if x is zero and 0 otherwise.

$$\text{operator!}(x) = \begin{cases} 0 & \text{if } x \neq 0 \\ 1 & \text{if } x = 0 \end{cases}$$

Available since 2.0

`int` **`operator!`**`(real x)`
Return 1 if x is zero and 0 otherwise.

$$\text{operator!}(x) = \begin{cases} 0 & \text{if } x \neq 0.0 \\ 1 & \text{if } x = 0.0 \end{cases}$$

deprecated; - use `operator==` instead.

Available since 2.0, deprecated in 2.31

`int` **`operator&&`**`(int x, int y)`

Return 1 if x is unequal to 0 and y is unequal to 0.

$$\text{operator\&\&}(x, y) = \begin{cases} 1 & \text{if } x \neq 0 \text{ and } y \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

Available since 2.0

`int` **`operator&&`**`(real x, real y)`
Return 1 if x is unequal to 0.0 and y is unequal to 0.0.

$$\text{operator\&\&}(x, y) = \begin{cases} 1 & \text{if } x \neq 0.0 \text{ and } y \neq 0.0 \\ 0 & \text{otherwise} \end{cases}$$

deprecated

Available since 2.0, deprecated in 2.31

`int` **`operator||`**`(int x, int y)`
Return 1 if x is unequal to 0 or y is unequal to 0.

$$\text{operator}\texttt{||}(x, y) = \begin{cases} 1 & \text{if } x \neq 0 \text{ or } y \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

Available since 2.0

`int` **`operator||`**`(real x, real y)`
Return 1 if x is unequal to 0.0 or y is unequal to 0.0.

$$\text{operator}\texttt{||}(x, y) = \begin{cases} 1 & \text{if } x \neq 0.0 \text{ or } y \neq 0.0 \\ 0 & \text{otherwise} \end{cases}$$

deprecated

Available since 2.0, deprecated in 2.31

Boolean operator short circuiting

Like in C++, the boolean operators `&&` and `||` and are implemented to short circuit directly to a return value after evaluating the first argument if it is sufficient to resolve the result. In evaluating `a || b`, if `a` evaluates to a value other than zero, the expression returns the value 1 without evaluating the expression `b`. Similarly, evaluating `a && b` first evaluates `a`, and if the result is zero, returns 0 without evaluating `b`.

Logical functions

The logical functions introduce conditional behavior functionally and are primarily provided for compatibility with BUGS and JAGS.

`real` **`step`**`(real x)`
Return 1 if x is positive and 0 otherwise.

$$\text{step}(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{otherwise} \end{cases}$$

Warning: `int_step(0)` and `int_step(NaN)` return 0 whereas `step(0)` and `step(NaN)` return 1.

The step function is often used in BUGS to perform conditional operations. For instance, `step(a-b)` evaluates to 1 if `a` is greater than `b` and evaluates to 0 otherwise. `step` is a step-like functions; see the warning in section step functions applied to expressions dependent on parameters.

Available since 2.0

`int` **`is_inf`**`(real x)`
Return 1 if x is infinite (positive or negative) and 0 otherwise.

Available since 2.5

`int` **`is_nan`**`(real x)`
Return 1 if x is NaN and 0 otherwise.

Available since 2.5

Care must be taken because both of these indicator functions are step-like and thus can cause discontinuities in gradients when applied to parameters; see section step-like functions for details.

3.6. Real-valued arithmetic operators

The arithmetic operators are presented using C++ notation. For instance `operator+(x,y)` refers to the binary addition operator and `operator-(x)` to the unary negation operator. In Stan programs, these are written using the usual infix and prefix notations as `x + y` and `-x`, respectively.

Binary infix operators

`real` **`operator+`**`(real x, real y)`
Return the sum of x and y.

$$(x + y) = \text{operator+}(x, y) = x + y$$

Available since 2.0

`real` **`operator-`**`(real x, real y)`
Return the difference between x and y.

$$(x - y) = \text{operator-}(x, y) = x - y$$

Available since 2.0

`real` **`operator*`**`(real x, real y)`
Return the product of x and y.

$$(x * y) = \text{operator*}(x, y) = xy$$

Available since 2.0

`real` **`operator/`**`(real x, real y)`
Return the quotient of x and y.

$$(x/y) = \text{operator/}(x, y) = \frac{x}{y}$$

Available since 2.0

`real` **`operator^`**`(real x, real y)`
Return x raised to the power of y.

$$(x^\wedge y) = \text{operator}^\wedge(x, y) = x^y$$

Available since 2.5

Unary prefix operators

`real` **`operator-`**`(real x)`
Return the negation of the subtrahend x.

$$\text{operator-}(x) = (-x)$$

Available since 2.0

`T` **`operator-`**`(T x)`
Vectorized version of `operator-`. If `T x` is a (possibly nested) array of reals, `-x` is the same shape array where each individual number is negated.

Available since 2.31

`real` **`operator+`**`(real x)`
Return the value of x.

$$\text{operator+}(x) = x$$

Available since 2.0

3.7. Step-like functions

Warning: *These functions can seriously hinder sampling and optimization efficiency for gradient-based methods (e.g., NUTS, HMC, BFGS) if applied to parameters (including transformed parameters and local variables in the transformed parameters or model block). The problem is that they break gradients due to discontinuities coupled with zero gradients elsewhere. They do not hinder sampling when used in the data, transformed data, or generated quantities blocks.*

Absolute value functions

`T` **`abs`**`(T x)`
The absolute value of x.

This function works elementwise over containers such as vectors. Given a type `T` which is `real` `vector`, `row_vector`, `matrix`, or an array of those types, `abs` returns the same type where each element has had its absolute value taken.

Available since 2.0, vectorized in 2.30

`real` **`fdim`**`(real x, real y)`
Return the positive difference between x and y, which is x - y if x is greater than y and 0 otherwise; see warning above.

$$\text{fdim}(x,y) = \begin{cases} x - y & \text{if } x \geq y \\ 0 & \text{otherwise} \end{cases}$$

Available since 2.0

`R` **`fdim`**`(T1 x, T2 y)`
Vectorized implementation of the `fdim` function

Available since 2.25

Bounds functions

`real` **`fmin`**`(real x, real y)`
Return the minimum of x and y; see warning above.

$$\text{fmin}(x,y) = \begin{cases} x & \text{if } x \leq y \\ y & \text{otherwise} \end{cases}$$

Available since 2.0

`R` **`fmin`**`(T1 x, T2 y)`
Vectorized implementation of the `fmin` function

Available since 2.25

`real` **`fmax`**`(real x, real y)`
Return the maximum of x and y; see warning above.

$$\text{fmax}(x, y) = \begin{cases} x & \text{if } x \geq y \\ y & \text{otherwise} \end{cases}$$

Available since 2.0

`R` **`fmax`**`(T1 x, T2 y)`
Vectorized implementation of the `fmax` function

Available since 2.25

Arithmetic functions

`real` **`fmod`**`(real x, real y)`
Return the real value remainder after dividing x by y; see warning above.

$$\text{fmod}(x, y) = x - \left\lfloor \frac{x}{y} \right\rfloor y$$

The operator $\lfloor u \rfloor$ is the floor operation; see below.

Available since 2.0

`R` **`fmod`**`(T1 x, T2 y)`
Vectorized implementation of the `fmod` function

Available since 2.25

Rounding functions

Warning: Rounding functions convert real values to integers. Because the output is an integer, any gradient information resulting from functions applied to the integer is not passed to the real value it was derived from. With MCMC sampling using HMC or NUTS, the MCMC acceptance procedure will correct for any error due to poor gradient calculations, but the result is likely to be reduced acceptance probabilities and less efficient sampling.

The rounding functions cannot be used as indices to arrays because they return real values. Stan may introduce integer-valued versions of these in the future, but as of now, there is no good workaround.

`R` **`floor`**`(T x)`
The floor of x, which is the largest integer less than or equal to x, converted to a real value; see warning at start of section step-like functions

Available since 2.0, vectorized in 2.13

`R` **`ceil`**`(T x)`
The ceiling of x, which is the smallest integer greater than or equal to x, converted to a real value; see warning at start of section step-like functions

Available since 2.0, vectorized in 2.13

`R` **`round`**`(T x)`
The nearest integer to x, converted to a real value; see warning at start of section step-like functions

Available since 2.0, vectorized in 2.13

`R` **`trunc`**`(T x)`
The integer nearest to but no larger in magnitude than x, converted to a double value; see warning at start of section step-like functions

Available since 2.0, vectorized in 2.13

3.8. Power and logarithm functions

`R` **`sqrt`**`(T x)`
The square root of x

Available since 2.0, vectorized in 2.13

`R` **`cbrt`**`(T x)`
The cube root of x

Available since 2.0, vectorized in 2.13

`R` **`square`**`(T x)`
The square of x

Available since 2.0, vectorized in 2.13

`R` **`exp`**`(T x)`
The natural exponential of x

Available since 2.0, vectorized in 2.13

`R` **`exp2`**`(T x)`
The base-2 exponential of x

Available since 2.0, vectorized in 2.13

`R` **`log`**`(T x)`
The natural logarithm of x

Available since 2.0, vectorized in 2.13

`R` **`log2`**`(T x)`
The base-2 logarithm of x

Available since 2.0, vectorized in 2.13

`R` **`log10`**`(T x)`
The base-10 logarithm of x

Available since 2.0, vectorized in 2.13

`real` **`pow`**`(real x, real y)`
Return x raised to the power of y.

$$\text{pow}(x, y) = x^y$$

Available since 2.0

`R` **`pow`**`(T1 x, T2 y)`
Vectorized implementation of the `pow` function

Available since 2.25

`R` **`inv`**`(T x)`
The inverse of x

Available since 2.0, vectorized in 2.13

`R` **`inv_sqrt`**`(T x)`
The inverse of the square root of x

Available since 2.0, vectorized in 2.13

`R` **`inv_square`**`(T x)`
The inverse of the square of x

Available since 2.0, vectorized in 2.13

3.9. Trigonometric functions

`real` **`hypot`**`(real x, real y)`
Return the length of the hypotenuse of a right triangle with sides of length x and y.

$$\text{hypot}(x, y) = \begin{cases} \sqrt{x^2 + y^2} & \text{if } x, y \geq 0 \\ \text{NaN} & \text{otherwise} \end{cases}$$

Available since 2.0

`R` **`hypot`**`(T1 x, T2 y)`
Vectorized implementation of the `hypot` function

Available since 2.25

`R` **`cos`**`(T x)`
The cosine of the angle x (in radians)

Available since 2.0, vectorized in 2.13

`R` **`sin`**`(T x)`
The sine of the angle x (in radians)

Available since 2.0, vectorized in 2.13

`R` **`tan`**`(T x)`
The tangent of the angle x (in radians)

Available since 2.0, vectorized in 2.13

`R` **`acos`**`(T x)`
The principal arc (inverse) cosine (in radians) of x

Available since 2.0, vectorized in 2.13

`R` **`asin`**`(T x)`
The principal arc (inverse) sine (in radians) of x

Available since 2.0

`R` **`atan`**`(T x)`
The principal arc (inverse) tangent (in radians) of x, with values from $-\pi/2$ to $\pi/2$

Available since 2.0, vectorized in 2.13

`R` **`atan2`**`(T y, T x)`
Return the principal arc (inverse) tangent (in radians) of y divided by x,

$$\text{atan2}(y, x) = \arctan\left(\frac{y}{x}\right)$$

Available since 2.0, vectorized in 2.34

3.10. Hyperbolic trigonometric functions

`R` **`cosh`**`(T x)`
The hyperbolic cosine of x (in radians)

Available since 2.0, vectorized in 2.13

`R` **`sinh`**`(T x)`
The hyperbolic sine of x (in radians)

Available since 2.0, vectorized in 2.13

`R` **`tanh`**`(T x)`
The hyperbolic tangent of x (in radians)

Available since 2.0, vectorized in 2.13

`R` **`acosh`**`(T x)`
The inverse hyperbolic cosine (in radians)

Available since 2.0, vectorized in 2.13

`R` **`asinh`**`(T x)`
The inverse hyperbolic cosine (in radians)

Available since 2.0, vectorized in 2.13

`R` **`atanh`**`(T x)`
The inverse hyperbolic tangent (in radians) of x

Available since 2.0, vectorized in 2.13

3.11. Link functions

The following functions are commonly used as link functions in generalized linear models. The function Φ is also commonly used as a link function (see section probability-related functions).

`R` **`logit`**`(T x)`
The log odds, or logit, function applied to x

Available since 2.0, vectorized in 2.13

`R` **`inv_logit`**`(T x)`
The logistic sigmoid function applied to x

Available since 2.0, vectorized in 2.13

`R` **`inv_cloglog`**`(T x)`
The inverse of the complementary log-log function applied to x

Available since 2.0, vectorized in 2.13

3.12. Probability-related functions

Normal cumulative distribution functions

The error function `erf` is related to the standard normal cumulative distribution function Φ by scaling. See section normal distribution for the general normal cumulative distribution function (and its complement).

`R` **`erf`**`(T x)`
The error function, also known as the Gauss error function, of x

Available since 2.0, vectorized in 2.13

`R` **`erfc`**`(T x)`
The complementary error function of x

Available since 2.0, vectorized in 2.13

`R` **`inv_erfc`**`(T x)`
The inverse of the complementary error function of x

Available since 2.29, vectorized in 2.29

`R` **`Phi`**`(T x)`
The standard normal cumulative distribution function of x

Available since 2.0, vectorized in 2.13

`R` **`inv_Phi`**`(T x)`
Return the value of the inverse standard normal cdf Φ^{-1} at the specified quantile `x`. The details of the algorithm can be found in (Wichura 1988). Quantile arguments below 1e-16 are untested; quantiles above 0.999999999 result in increasingly large errors.

Available since 2.0, vectorized in 2.13

`R` **`Phi_approx`**`(T x)`
The fast approximation of the unit (may replace `Phi` for probit regression with maximum absolute error of 0.00014, see (Bowling et al. 2009) for details)

Available since 2.0, vectorized in 2.13

Other probability-related functions

`real` **`binary_log_loss`**`(int y, real y_hat)`
Return the log loss function for for predicting $\hat{y} \in [0,1]$ for boolean outcome $y \in \{0,1\}$.

$$\text{binary\_log\_loss}(y, \hat{y}) = \begin{cases} -\log \hat{y} & \text{if } y = 1 \\ -\log(1-\hat{y}) & \text{otherwise} \end{cases}$$

Available since 2.0

`R` **`binary_log_loss`**`(T1 x, T2 y)`
Vectorized implementation of the `binary_log_loss` function

Available since 2.25

`real` **`owens_t`**`(real h, real a)`
Return the Owen's T function for the probability of the event $X > h$ and $0 < Y < aX$ where X and Y are independent standard normal random variables.

$$\text{owens\_t}(h, a) = \frac{1}{2\pi} \int_0^a \frac{\exp(-\frac{1}{2}h^2(1+x^2))}{1+x^2} dx$$

Available since 2.25

`R` **`owens_t`**`(T1 x, T2 y)`
Vectorized implementation of the `owens_t` function

Available since 2.25

3.13. Combinatorial functions

`real` **`beta`**`(real alpha, real beta)`
Return the beta function applied to alpha and beta. The beta function, $\text{B}(\alpha, \beta)$, computes the normalizing constant for the beta distribution, and is defined for $\alpha > 0$ and $\beta > 0$. See section appendix for definition of $\text{B}(\alpha, \beta)$.

Available since 2.25

`R` **`beta`**`(T1 x, T2 y)`
Vectorized implementation of the `beta` function

Available since 2.25

`real` **`inc_beta`**`(real alpha, real beta, real x)`
Return the regularized incomplete beta function up to x applied to alpha and beta. See section appendix for a definition.

Available since 2.10

`real` **`inv_inc_beta`**`(real alpha, real beta, real p)`
Return the inverse of the regularized incomplete beta function. The return value `x` is the value that solves `p = inc_beta(alpha, beta, x)`. See section appendix for a definition of the `inc_beta`.

Available since 2.30

`real` **`lbeta`**`(real alpha, real beta)`
Return the natural logarithm of the beta function applied to alpha and beta. The beta function, $\mathrm{B}(\alpha, \beta)$, computes the normalizing constant for the beta distribution, and is defined for $\alpha > 0$ and $\beta > 0$.

$$\mathrm{lbeta}(\alpha, \beta) = \log \Gamma(a) + \log \Gamma(b) - \log \Gamma(a + b)$$

See section appendix for definition of $\mathrm{B}(\alpha, \beta)$.

Available since 2.0

`R` **`lbeta`**`(T1 x, T2 y)`
Vectorized implementation of the `lbeta` function

Available since 2.25

`R` **`tgamma`**`(T x)`
The gamma function applied to x. The gamma function is the generalization of the factorial function to continuous variables, defined so that $\Gamma(n + 1) = n!$. See for a full definition of $\Gamma(x)$. The function is defined for positive numbers and non-integral negative numbers,

Available since 2.0, vectorized in 2.13

`R` **`lgamma`**`(T x)`
The natural logarithm of the gamma function applied to x,

Available since 2.0, vectorized in 2.15

`R` **`digamma`**`(T x)`
The digamma function applied to x. The digamma function is the derivative of

the natural logarithm of the Gamma function. The function is defined for positive numbers and non-integral negative numbers

Available since 2.0, vectorized in 2.13

`R` **`trigamma`**`(T x)`
The trigamma function applied to x. The trigamma function is the second derivative of the natural logarithm of the Gamma function

Available since 2.0, vectorized in 2.13

`real` **`lmgamma`**`(int n, real x)`
Return the natural logarithm of the multivariate gamma function Γ_n with n dimensions applied to x.

$$\text{lmgamma}(n, x) = \begin{cases} \frac{n(n-1)}{4} \log \pi + \sum_{j=1}^{n} \log \Gamma\left(x + \frac{1-j}{2}\right) & \text{if } x \notin \{\ldots, -3, -2, -1, 0\} \\ \text{error} & \text{otherwise} \end{cases}$$

Available since 2.0

`R` **`lmgamma`**`(T1 x, T2 y)`
Vectorized implementation of the `lmgamma` function

Available since 2.25

`real` **`gamma_p`**`(real a, real z)`
Return the normalized lower incomplete gamma function of a and z defined for positive a and nonnegative z.

$$\text{gamma\_p}(a, z) = \begin{cases} \frac{1}{\Gamma(a)} \int_0^z t^{a-1} e^{-t} dt & \text{if } a > 0, z \geq 0 \\ \text{error} & \text{otherwise} \end{cases}$$

Available since 2.0

`R` **`gamma_p`**`(T1 x, T2 y)`
Vectorized implementation of the `gamma_p` function

Available since 2.25

`real` **`gamma_q`**`(real a, real z)`
Return the normalized upper incomplete gamma function of a and z defined for

positive a and nonnegative z.

$$\text{gamma\_q}(a,z) = \begin{cases} \frac{1}{\Gamma(a)} \int_z^\infty t^{a-1} e^{-t} dt & \text{if } a > 0, z \geq 0 \\ \text{error} & \text{otherwise} \end{cases}$$

Available since 2.0

`R` **`gamma_q`**`(T1 x, T2 y)`
Vectorized implementation of the `gamma_q` function

Available since 2.25

`int` **`choose`**`(int x, int y)`
Return the binomial coefficient of x and y. For non-negative integer inputs, the binomial coefficient function is written as $\binom{x}{y}$ and pronounced "x choose y." In its the antilog of the `lchoose` function but returns an integer rather than a real number with no non-zero decimal places. For $0 \leq y \leq x$, the binomial coefficient function can be defined via the factorial function

$$\text{choose}(x,y) = \frac{x!}{(y!)\,(x-y)!}.$$

Available since 2.14

`R` **`choose`**`(T1 x, T2 y)`
Vectorized implementation of the `choose` function

Available since 2.25

`real` **`bessel_first_kind`**`(int v, real x)`
Return the Bessel function of the first kind with order v applied to x.

$$\text{bessel\_first\_kind}(v,x) = J_v(x),$$

where

$$J_v(x) = \left(\frac{1}{2}x\right)^v \sum_{k=0}^{\infty} \frac{\left(-\frac{1}{4}x^2\right)^k}{k!\,\Gamma(v+k+1)}$$

Available since 2.5

`R` **`bessel_first_kind`**`(T1 x, T2 y)`
Vectorized implementation of the `bessel_first_kind` function

Available since 2.25

`real` **`bessel_second_kind`**`(int v, real x)`
Return the Bessel function of the second kind with order v applied to x defined for positive x and v. For $x, v > 0$,

$$\text{bessel\_second\_kind}(v, x) = \begin{cases} Y_v(x) & \text{if } x > 0 \\ \text{error} & \text{otherwise} \end{cases}$$

where

$$Y_v(x) = \frac{J_v(x)\cos(v\pi) - J_{-v}(x)}{\sin(v\pi)}$$

Available since 2.5

`R` **`bessel_second_kind`**`(T1 x, T2 y)`
Vectorized implementation of the `bessel_second_kind` function

Available since 2.25

`real` **`modified_bessel_first_kind`**`(int v, real z)`
Return the modified Bessel function of the first kind with order v applied to z defined for all z and integer v.

$$\text{modified\_bessel\_first\_kind}(v, z) = I_v(z)$$

where

$$I_v(z) = \left(\frac{1}{2}z\right)^v \sum_{k=0}^{\infty} \frac{\left(\frac{1}{4}z^2\right)^k}{k!\Gamma(v + k + 1)}$$

Available since 2.1

`R` **`modified_bessel_first_kind`**`(T1 x, T2 y)`
Vectorized implementation of the `modified_bessel_first_kind` function

Available since 2.25

`real` **`log_modified_bessel_first_kind`**`(real v, real z)`
Return the log of the modified Bessel function of the first kind. v does not have to be an integer.

Available since 2.26

`R` **`log_modified_bessel_first_kind`**`(T1 x, T2 y)`
Vectorized implementation of the `log_modified_bessel_first_kind` function

Available since 2.26

`real` **`modified_bessel_second_kind`**`(int v, real z)`
Return the modified Bessel function of the second kind with order v applied to z defined for positive z and integer v.

$$\text{modified\_bessel\_second\_kind}(v,z) = \begin{cases} K_v(z) & \text{if } z > 0 \\ \text{error} & \text{if } z \le 0 \end{cases}$$

where

$$K_v(z) = \frac{\pi}{2} \cdot \frac{I_{-v}(z) - I_v(z)}{\sin(v\pi)}$$

Available since 2.1

`R` **`modified_bessel_second_kind`**`(T1 x, T2 y)`
Vectorized implementation of the `modified_bessel_second_kind` function

Available since 2.25

`real` **`falling_factorial`**`(real x, real n)`
Return the falling factorial of x with power n defined for positive x and real n.

$$\text{falling\_factorial}(x,n) = \begin{cases} (x)_n & \text{if } x > 0 \\ \text{error} & \text{if } x \le 0 \end{cases}$$

where

$$(x)_n = \frac{\Gamma(x+1)}{\Gamma(x-n+1)}$$

Available since 2.0

`R` **`falling_factorial`**`(T1 x, T2 y)`
Vectorized implementation of the `falling_factorial` function

Available since 2.25

`real` **`lchoose`**`(real x, real y)`
Return the natural logarithm of the generalized binomial coefficient of x and y. For non-negative integer inputs, the binomial coefficient function is written as $\binom{x}{y}$ and pronounced “x choose y.” This function generalizes to real numbers using the gamma function. For $0 \le y \le x$,

$$\text{binomial\_coefficient\_log}(x,y) = \log\Gamma(x+1) - \log\Gamma(y+1) - \log\Gamma(x-y+1).$$

Available since 2.10

`R` **`lchoose`**`(T1 x, T2 y)`
Vectorized implementation of the `lchoose` function

Available since 2.29

`real` **`log_falling_factorial`**`(real x, real n)`
Return the log of the falling factorial of x with power n defined for positive x and real n.

$$\text{log\_falling\_factorial}(x, n) = \begin{cases} \log(x)_n & \text{if } x > 0 \\ \text{error} & \text{if } x \leq 0 \end{cases}$$

Available since 2.0

`real` **`rising_factorial`**`(real x, int n)`
Return the rising factorial of x with power n defined for positive x and integer n.

$$\text{rising\_factorial}(x, n) = \begin{cases} x^{(n)} & \text{if } x > 0 \\ \text{error} & \text{if } x \leq 0 \end{cases}$$

where

$$x^{(n)} = \frac{\Gamma(x+n)}{\Gamma(x)}$$

Available since 2.20

`R` **`rising_factorial`**`(T1 x, T2 y)`
Vectorized implementation of the `rising_factorial` function

Available since 2.25

`real` **`log_rising_factorial`**`(real x, real n)`
Return the log of the rising factorial of x with power n defined for positive x and real n.

$$\text{log\_rising\_factorial}(x, n) = \begin{cases} \log x^{(n)} & \text{if } x > 0 \\ \text{error} & \text{if } x \leq 0 \end{cases}$$

Available since 2.0

`R` **`log_rising_factorial`**`(T1 x, T2 y)`
Vectorized implementation of the `log_rising_factorial` function

Available since 2.25

3.14. Composed functions

The functions in this section are equivalent in theory to combinations of other functions. In practice, they are implemented to be more efficient and more numerically stable than defining them directly using more basic Stan functions.

`R` **`expm1`**`(T x)`
The natural exponential of x minus 1

Available since 2.0, vectorized in 2.13

`real` **`fma`**`(real x, real y, real z)`
Return z plus the result of x multiplied by y.

$$\text{fma}(x, y, z) = (x \times y) + z$$

Available since 2.0

`real` **`ldexp`**`(real x, int y)`
Return the product of x and two raised to the y power.

$$\text{ldexp}(x, y) = x2^y$$

Available since 2.25

`R` **`ldexp`**`(T1 x, T2 y)`
Vectorized implementation of the `ldexp` function

Available since 2.25

`real` **`lmultiply`**`(real x, real y)`
Return the product of x and the natural logarithm of y.

$$\text{lmultiply}(x, y) = \begin{cases} 0 & \text{if } x = y = 0 \\ x \log y & \text{if } x, y \neq 0 \\ \text{NaN} & \text{otherwise} \end{cases}$$

Available since 2.10

`R` **`lmultiply`**`(T1 x, T2 y)`
Vectorized implementation of the `lmultiply` function

Available since 2.25

`R` **`log1p`**`(T x)`
The natural logarithm of 1 plus x

Available since 2.0, vectorized in 2.13

`R` **`log1m`**`(T x)`
The natural logarithm of 1 minus x

Available since 2.0, vectorized in 2.13

`R` **`log1p_exp`**`(T x)`
The natural logarithm of one plus the natural exponentiation of x

Available since 2.0, vectorized in 2.13

`R` **`log1m_exp`**`(T x)`
The logarithm of one minus the natural exponentiation of x

Available since 2.0, vectorized in 2.13

`real` **`log_diff_exp`**`(real x, real y)`
Return the natural logarithm of the difference of the natural exponentiation of x and the natural exponentiation of y.

$$\text{log\_diff\_exp}(x, y) = \begin{cases} \log(\exp(x) - \exp(y)) & \text{if } x > y \\ \text{NaN} & \text{otherwise} \end{cases}$$

Available since 2.0

`R` **`log_diff_exp`**`(T1 x, T2 y)`
Vectorized implementation of the `log_diff_exp` function

Available since 2.25

`real` **`log_mix`**`(real theta, real lp1, real lp2)`
Return the log mixture of the log densities lp1 and lp2 with mixing proportion theta, defined by

$$\begin{aligned} \text{log\_mix}(\theta, \lambda_1, \lambda_2) &= \log(\theta \exp(\lambda_1) + (1 - \theta) \exp(\lambda_2)) \\ &= \text{log\_sum\_exp}(\log(\theta) + \lambda_1, \ \log(1 - \theta) + \lambda_2) \,. \end{aligned}$$

Available since 2.6

`R` **`log_mix`**`(T1 theta, T2 lp1, T3 lp2)`
Vectorized implementation of the `log_mix` function

Available since 2.26

R **log_sum_exp**(T1 x, T2 y)
Return the natural logarithm of the sum of the natural exponentiation of x and the natural exponentiation of y.

$$\text{log\_sum\_exp}(x, y) = \log(\exp(x) + \exp(y))$$

Available since 2.0, vectorized in 2.33

R **log_inv_logit**(T x)
The natural logarithm of the inverse logit function of x

Available since 2.0, vectorized in 2.13

R **log_inv_logit_diff**(T1 x, T2 y)
The natural logarithm of the difference of the inverse logit function of x and the inverse logit function of y

Available since 2.25

R **log1m_inv_logit**(T x)
The natural logarithm of 1 minus the inverse logit function of x

Available since 2.0, vectorized in 2.13

3.15. Special functions

R **lambert_w0**(T x)
Implementation of the W_0 branch of the Lambert W function, i.e., solution to the function $W_0(x)\exp^{W_0(x)} = x$

Available since 2.25

R **lambert_wm1**(T x)
Implementation of the W_{-1} branch of the Lambert W function, i.e., solution to the function $W_{-1}(x)\exp^{W_{-1}(x)} = x$

Available since 2.25

4. Complex-Valued Basic Functions

This chapter describes built-in functions that operate on complex numbers, either as an argument type or a return type. This includes the arithmetic operators generalized to complex numbers.

4.1. Complex assignment and promotion

Just as integers may be assigned to real variables, real variables may be assigned to complex numbers, with the result being a zero imaginary component.

```
int n = 5;         // n = 5
real x = a;        // x = 5.0
complex z1 = n;  // z = 5.0 + 0.0i
complex z2 = x;  // z = 5.0 + 0.0i
```

Complex function arguments

Function arguments of type `int` or `real` may be promoted to type `complex`. The complex version of functions in this chapter are only used if one of the arguments is complex. For example, if `z` is complex, then `pow(z, 2)` will call the complex version of the power function and the integer 2 will be promoted to a complex number with a real component of 2 and an imaginary component of 0. The same goes for binary operators like addition and subtraction, where `z + 2` will be legal and produce a complex result. Functions such as `arg` and `conj` that are only available for complex numbers can accept integer or real arguments, promoting them to `complex` before applying the function.

4.2. Complex constructors and accessors

Complex constructors

Variables and constants of type `complex` are constructed from zero, one, or two real numbers.

```
complex z1 = to_complex();          // z1 = 0.0 + 0.0i
real re = -2.9;
complex z2 = to_complex(re);       // z2 = -2.9 + 0.0i
real im = 1.3;
complex z3 = to_complex(re, im);  // z3 = -2.9 + 1.3i
```

`complex` **`to_complex`**`()`
Return complex number with real part 0.0 and imaginary part 0.0.

Available since 2.28

`complex` **`to_complex`**`(real re)`
Return complex number with real part `re` and imaginary part 0.0.

Available since 2.28

`complex` **`to_complex`**`(real re, real im)`
Return complex number with real part `re` and imaginary part `im`.

Available since 2.28

`Z` **`to_complex`**`(T1 re, T2 im)`
Vectorized implementation of the `to_complex` function.

`T1` and `T2` can either be real containers of the same size, or a real container and a real, in which case the real value is used for the corresponding component in all elements of the output.

Available since 2.30

Complex accessors

Given a complex number, its real and imaginary parts can be extracted with the following functions.

`real` **`get_real`**`(complex z)`
Return the real part of the complex number `z`.

Available since 2.28

`real` **`get_imag`**`(complex z)`
Return the imaginary part of the complex number `z`.

Available since 2.28

4.3. Complex arithmetic operators

The arithmetic operators have the same precedence for complex and real arguments. The complex form of an operator will be selected if at least one of its argument is of type `complex`. If there are two arguments and only one is of type `complex`, then the other will be promoted to type `complex` before performing the operation.

Unary operators

`complex` **`operator+`**`(complex z)`
Return the complex argument z,

$$+z = z.$$

Available since 2.28

`complex` **`operator-`**`(complex z)`
Return the negation of the complex argument z, which for $z = x + yi$ is

$$-z = -x - yi.$$

Available since 2.28

`T` **`operator-`**`(T x)`
Vectorized version of `operator-`. If `T x` is a (possibly nested) array of complex numbers, `-x` is the same shape array where each individual value is negated.

Available since 2.31

Binary operators

`complex` **`operator+`**`(complex x, complex y)`
Return the sum of x and y,

$$(x + y) = \text{operator+}(x, y) = x + y.$$

Available since 2.28

`complex` **`operator-`**`(complex x, complex y)`
Return the difference between x and y,

$$(x - y) = \text{operator-}(x, y) = x - y.$$

Available since 2.28

`complex` **`operator*`**`(complex x, complex y)`
Return the product of x and y,

$$(x * y) = \text{operator*}(x, y) = x \times y.$$

Available since 2.28

`complex` **`operator/`**`(complex x, complex y)`
Return the quotient of x and y,

$$(x/y) = \text{operator}/(x, y) = \frac{x}{y}$$

Available since 2.28

`complex` **`operator^`**`(complex x, complex y)`
Return x raised to the power of y,

$$(x^\wedge y) = \text{operator}^\wedge(x, y) = \exp(y \, \log(x)).$$

Available since 2.28

4.4. Complex comparison operators

Complex numbers are equal if and only if both their real and imaginary components are equal. That is, the conditional

```
z1 == z2
```

is equivalent to

```
get_real(z1) == get_real(z2) && get_imag(z1) == get_imag(z2)
```

As with other complex functions, if one of the arguments is of type `real` or `int`, it will be promoted to type `complex` before comparison. For example, if `z` is of type `complex`, then `z == 0` will be true if `z` has real component equal to 0.0 and complex component equal to 0.0.

Warning: As with real values, it is usually a mistake to compare complex numbers for equality because their parts are implemented using floating-point arithmetic, which suffers from precision errors, rendering algebraically equivalent expressions not equal after evaluation.

`int` **`operator==`**`(complex x, complex y)`
Return 1 if x is equal to y and 0 otherwise,

$$(x == y) \;=\; \text{operator==}(x, y) \;=\; \begin{cases} 1 & \text{if } x = y, \text{ and} \\ 0 & \text{otherwise.} \end{cases}$$

Available since 2.28

`int` **`operator!=`**`(complex x, complex y)`
Return 1 if x is not equal to y and 0 otherwise,

$$(x \mathrel{!=} y) \;=\; \text{operator!=}(x, y) \;=\; \begin{cases} 1 & \text{if } x \neq y, \text{ and} \\ 0 & \text{otherwise.} \end{cases}$$

Available since 2.28

4.5. Complex (compound) assignment operators

The assignment operator only serves as a component in the assignment statement and is thus not technically a function in the Stan language. With that caveat, it is documented here for completeness.

Assignment of complex numbers works elementwise. If an expression of type `int` or `real` is assigned to a complex number, it will be promoted before assignment as if calling `to_complex()`, so that the imaginary component is 0.0.

`void` **`operator=`**`(complex x, complex y)`
`y = x;` assigns a (copy of) the value of `y` to `x`.

Available since 2.28

`void` **`operator+=`**`(complex x, complex y)`
`x += y;` is equivalent to `x = x + y;`.

Available since 2.28

`void` **`operator-=`**`(complex x, complex y)`
`x -= y;` is equivalent to `x = x - y;`.

Available since 2.28

`void` **`operator*=`**`(complex x, complex y)`
`x *= y;` is equivalent to `x = x * y;`.

Available since 2.28

`void` **`operator/=`**`(complex x, complex y)`
`x /= y;` is equivalent to `x = x / y;`.

Available since 2.28

4.6. Complex special functions

The following functions are specific to complex numbers other than absolute value, which has a specific meaning for complex numbers.

`real` **`abs`**`(complex z)`
Return the absolute value of z, also known as the modulus or magnitude, which for $z = x + yi$ is

$$\text{abs}(z) = \sqrt{x^2 + y^2}.$$

This function works elementwise over containers, returning the same shape and kind of the input container but holding reals. For example, a `complex_vector[n]` input will return a `vector[n]` output, with each element transformed by the above equation.

Available since 2.28, vectorized in 2.30

`real` **`arg`**`(complex z)`
Return the phase angle (in radians) of z, which for $z = x + yi$ is

$$\arg(z) = \text{atan2}(y, x) = \text{atan}(y/x).$$

Available since 2.28

`real` **`norm`**`(complex z)`
Return the Euclidean norm of z, which is its absolute value squared, and which for $z = x + yi$ is

$$\text{norm}(z) = \text{abs}^2(z) = x^2 + y^2.$$

Available since 2.28

`complex` **`conj`**`(complex z)`
Return the complex conjugate of z, which negates the imaginary component, so that if $z = x + yi$,

$$\text{conj}(z) = x - yi.$$

Available since 2.28

`Z` **`conj`**`(Z z)`
Vectorized version of `conj`. This will apply the `conj` function to each element of a complex array, vector, or matrix.

Available since 2.31

`complex` **`proj`**`(complex z)`
Return the projection of z onto the Riemann sphere, which for $z = x + yi$ is

$$\text{proj}(z) = \begin{cases} z & \text{if } z \text{ is finite, and} \\ 0 + \text{sign}(y)i & \text{otherwise,} \end{cases}$$

where $\text{sign}(y)$ is -1 if y is negative and 1 otherwise.

Available since 2.28

`complex` **`polar`**`(real r, real theta)`
Return the complex number with magnitude (absolute value) `r` and phase angle `theta`.

Available since 2.28

4.7. Complex exponential and power functions

The exponential, log, and power functions may be supplied with complex arguments with specialized meanings that generalize their real counterparts. These versions are only called when the argument is complex.

`complex` **`exp`**`(complex z)`
Return the complex natural exponential of `z`, which for $z = x + yi$ is

$$\exp z = \exp(x)\text{cis}(y) = \exp(x)(\cos(y) + i\sin(y)).$$

Available since 2.28

`complex` **`log`**`(complex z)`
Return the complex natural logarithm of `z`, which for $z = \text{polar}(r, \theta)$ is

$$\log z = \log r + \theta i.$$

Available since 2.28

`complex` **`log10`**`(complex z)`
Return the complex common logarithm of `z`,

$$\log_{10} z = \frac{\log z}{\log 10}.$$

Available since 2.28

`complex` **`pow`**`(complex x, complex y)`
Return x raised to the power of y,

$$\text{pow}(x, y) = \exp(y \log(x)).$$

Available since 2.28

`Z` **`pow`**`(T1 x, T2 y)`
Vectorized implementation of the `pow` function

Available since 2.30

`complex` **`sqrt`**`(complex x)`
Return the complex square root of x with branch cut along the negative real axis. For finite inputs, the result will be in the right half-plane.

Available since 2.28

4.8. Complex trigonometric functions

The standard trigonometric functions are supported for complex numbers.

`complex` **`cos`**`(complex z)`
Return the complex cosine of z, which is

$$\cos(z) = \cosh(z\,i) = \frac{\exp(z\,i) + \exp(-z\,i)}{2}.$$

Available since 2.28

`complex` **`sin`**`(complex z)`
Return the complex sine of z,

$$\sin(z) = -\sinh(z\,i)\,i = \frac{\exp(z\,i) - \exp(-z\,i)}{2\,i}.$$

Available since 2.28

`complex` **`tan`**`(complex z)`
Return the complex tangent of z,

$$\tan(z) = -\tanh(z\,i)\,i = \frac{(\exp(-z\,i) - \exp(z\,i))\,i}{\exp(-z\,i) + \exp(z\,i)}.$$

Available since 2.28

`complex` **`acos`**`(complex z)`
Return the complex arc (inverse) cosine of z,

$$\text{acos}(z) = \frac{1}{2}\pi + \log(z\,i + \sqrt{1 - z^2})\,i.$$

Available since 2.28

`complex` **`asin`**`(complex z)`
Return the complex arc (inverse) sine of z,

$$\mathrm{asin}(z) = -\log(z\,i + \sqrt{1 - z^2})\,i.$$

Available since 2.28

`complex` **`atan`**`(complex z)`
Return the complex arc (inverse) tangent of z,

$$\mathrm{atan}(z) = -\frac{1}{2}(\log(1 - z\,i) - \log(1 + z\,i))\,i.$$

Available since 2.28

4.9. Complex hyperbolic trigonometric functions

The standard hyperbolic trigonometric functions are supported for complex numbers.

`complex` **`cosh`**`(complex z)`
Return the complex hyperbolic cosine of z,

$$\cosh(z) = \frac{\exp(z) + \exp(-z)}{2}.$$

Available since 2.28

`complex` **`sinh`**`(complex z)`
Return the complex hyperbolic sine of z,

$$\sinh(z) = \frac{\exp(z) - \exp(-z)}{2}.$$

Available since 2.28

`complex` **`tanh`**`(complex z)`
Return the complex hyperbolic tangent of z,

$$\tanh(z) \;=\; \frac{\sinh(z)}{\cosh(z)} \;=\; \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}.$$

Available since 2.28

`complex` **`acosh`**`(complex z)`
Return the complex hyperbolic arc (inverse) cosine of z,

$$\text{acosh}(z) = \log(z + \sqrt{(z+1)(z-1)}).$$

Available since 2.28

`complex` **`asinh`**`(complex z)`
Return the complex hyperbolic arc (inverse) sine of z,

$$\text{asinh}(z) = \log(z + \sqrt{1+z^2}).$$

Available since 2.28

`complex` **`atanh`**`(complex z)`
Return the complex hyperbolic arc (inverse) tangent of z,

$$\text{atanh}(z) = \frac{\log(1+z) - \log(1-z)}{2}.$$

Available since 2.28

5. Array Operations

5.1. Reductions

The following operations take arrays as input and produce single output values. The boundary values for size 0 arrays are the unit with respect to the combination operation (min, max, sum, or product).

Minimum and maximum

`real` **`min`**`(array[] real x)`
The minimum value in x, or $+\infty$ if x is size 0.

Available since 2.0

`int` **`min`**`(array[] int x)`
The minimum value in x, or error if x is size 0.

Available since 2.0

`real` **`max`**`(array[] real x)`
The maximum value in x, or $-\infty$ if x is size 0.

Available since 2.0

`int` **`max`**`(array[] int x)`
The maximum value in x, or error if x is size 0.

Available since 2.0

Sum, product, and log sum of exp

`int` **`sum`**`(array[] int x)`
The sum of the elements in x, or 0 if the array is empty.

Available since 2.1

`real` **`sum`**`(array[] real x)`
The sum of the elements in x; see definition above.

Available since 2.0

`complex` **`sum`**`(array[] complex x)`
The sum of the elements in x; see definition above.

Available since 2.30

`real` **`prod`**`(array[] real x)`
The product of the elements in x, or 1 if x is size 0.

Available since 2.0

`real` **`prod`**`(array[] int x)`
The product of the elements in x,

$$\text{product}(x) = \begin{cases} \prod_{n=1}^{N} x_n & \text{if} N > 0 \\ 1 & \text{if} N = 0 \end{cases}$$

Available since 2.0

`real` **`log_sum_exp`**`(array[] real x)`
The natural logarithm of the sum of the exponentials of the elements in x, or $-\infty$ if the array is empty.

Available since 2.0

Sample mean, variance, and standard deviation

The sample mean, variance, and standard deviation are calculated in the usual way. For i.i.d. draws from a distribution of finite mean, the sample mean is an unbiased estimate of the mean of the distribution. Similarly, for i.i.d. draws from a distribution of finite variance, the sample variance is an unbiased estimate of the variance.[1] The sample deviation is defined as the square root of the sample deviation, but is not unbiased.

`real` **`mean`**`(array[] real x)`
The sample mean of the elements in x. For an array x of size $N > 0$,

$$\text{mean}(x) \; = \; \bar{x} \; = \; \frac{1}{N} \sum_{n=1}^{N} x_n.$$

It is an error to the call the mean function with an array of size 0.

Available since 2.0

[1] Dividing by N rather than $(N - 1)$ produces a maximum likelihood estimate of variance, which is biased to underestimate variance.

`real` **`variance`**`(array[] real x)`
The sample variance of the elements in x. For $N > 0$,

$$\text{variance}(x) \;=\; \begin{cases} \frac{1}{N-1} \sum_{n=1}^{N} (x_n - \bar{x})^2 & \text{if } N > 1 \\ 0 & \text{if } N = 1 \end{cases}$$

It is an error to call the `variance` function with an array of size 0.

Available since 2.0

`real` **`sd`**`(array[] real x)`
The sample standard deviation of elements in x.

$$\text{sd}(x) = \begin{cases} \sqrt{\text{variance}(x)} & \text{if } N > 1 \\ 0 & \text{if } N = 0 \end{cases}$$

It is an error to call the `sd` function with an array of size 0.

Available since 2.0

Norms

`real` **`norm1`**`(vector x)`
The L1 norm of x, defined by

$$\text{norm1}(x) \;=\; \textstyle\sum_{n=1}^{N} (|x_n|)$$

where `N` is the size of x.

Available since 2.30

`real` **`norm1`**`(row_vector x)`
The L1 norm of x

Available since 2.30

`real` **`norm1`**`(array[] real x)`
The L1 norm of x

Available since 2.30

`real` **`norm2`**`(vector x)`
The L2 norm of x, defined by

$$\text{norm2}(x) \;=\; \sqrt{\textstyle\sum_{n=1}^{N} (x_n)^2}$$

where `N` is the size of x

Available since 2.30

`real` **`norm2`**`(row_vector x)`
The L2 norm of x

Available since 2.30

`real` **`norm2`**`(array[] real x)`
The L2 norm of x

Available since 2.30

Euclidean distance and squared distance

`real` **`distance`**`(vector x, vector y)`
The Euclidean distance between x and y, defined by

$$\text{distance}(x, y) = \sqrt{\textstyle\sum_{n=1}^{N}(x_n - y_n)^2}$$

where `N` is the size of x and y. It is an error to call `distance` with arguments of unequal size.

Available since 2.2

`real` **`distance`**`(vector x, row_vector y)`
The Euclidean distance between x and y

Available since 2.2

`real` **`distance`**`(row_vector x, vector y)`
The Euclidean distance between x and y

Available since 2.2

`real` **`distance`**`(row_vector x, row_vector y)`
The Euclidean distance between x and y

Available since 2.2

`real` **`squared_distance`**`(vector x, vector y)`
The squared Euclidean distance between x and y, defined by

$$\text{squared\_distance}(x, y) = \text{distance}(x, y)^2 = \textstyle\sum_{n=1}^{N}(x_n - y_n)^2,$$

where `N` is the size of x and y. It is an error to call `squared_distance` with arguments of unequal size.

Available since 2.7

`real` **`squared_distance`**`(vector x, row_vector y)`
The squared Euclidean distance between x and y

Available since 2.26

`real` **`squared_distance`**`(row_vector x, vector y)`
The squared Euclidean distance between x and y

Available since 2.26

`real` **`squared_distance`**`(row_vector x, row_vector y)`
The Euclidean distance between x and y

Available since 2.26

Quantile

Produces sample quantiles corresponding to the given probabilities. The smallest observation corresponds to a probability of 0 and the largest to a probability of 1.

Implements algorithm 7 from Hyndman, R. J. and Fan, Y., Sample quantiles in Statistical Packages (R's default quantile function).

`real` **`quantile`**`(data array[] real x, data real p)`
The p-th quantile of x

Available since 2.27

`array[] real` **`quantile`**`(data array[] real x, data array[] real p)`
An array containing the quantiles of x given by the array of probabilities p

Available since 2.27

5.2. Array size and dimension function

The size of an array or matrix can be obtained using the `dims()` function. The `dims()` function is defined to take an argument consisting of any variable with up to 8 array dimensions (and up to 2 additional matrix dimensions) and returns an array of integers with the dimensions. For example, if two variables are declared as follows,

```
array[7, 8, 9] real x;
array[7] matrix[8, 9] y;
```

then calling `dims(x)` or `dims(y)` returns an integer array of size 3 containing the elements 7, 8, and 9 in that order.

The `size()` function extracts the number of elements in an array. This is just the top-level elements, so if the array is declared as

```
array[M, N] real a;
```

the size of `a` is `M`.

The function `num_elements`, on the other hand, measures all of the elements, so that the array `a` above has $M \times N$ elements.

The specialized functions `rows()` and `cols()` should be used to extract the dimensions of vectors and matrices.

`array[] int` **`dims`**`(T x)`
Return an integer array containing the dimensions of `x`; the type of the argument `T` can be any Stan type with up to 8 array dimensions.

Available since 2.0

`int` **`num_elements`**`(array[] T x)`
Return the total number of elements in the array `x` including all elements in contained arrays, vectors, and matrices. `T` can be any array type. For example, if `x` is of type `array[4, 3] real` then `num_elements(x)` is 12, and if `y` is declared as `array[5] matrix[3, 4] y`, then `size(y)` evaluates to 60.

Available since 2.5

`int` **`size`**`(array[] T x)`
Return the number of elements in the array `x`; the type of the array `T` can be any type, but the size is just the size of the top level array, not the total number of elements contained. For example, if `x` is of type `array[4, 3] real` then `size(x)` is 4.

Available since 2.0

5.3. Array broadcasting

The following operations create arrays by repeating elements to fill an array of a specified size. These operations work for all input types T, including reals, integers, vectors, row vectors, matrices, or arrays.

`array[] T` **`rep_array`**`(T x, int n)`
Return the `n` array with every entry assigned to `x`.

Available since 2.0

`array [,] T` **`rep_array`**`(T x, int m, int n)`
Return the m by n array with every entry assigned to x.

Available since 2.0

`array[,,] T` **`rep_array`**`(T x, int k, int m, int n)`
Return the k by m by n array with every entry assigned to x.

Available since 2.0

For example, `rep_array(1.0,5)` produces a real array (type `array[] real`) of size 5 with all values set to 1.0. On the other hand, `rep_array(1,5)` produces an integer array (type `array[] int`) of size 5 with all values set to 1. This distinction is important because it is not possible to assign an integer array to a real array. For example, the following example contrasts legal with illegal array creation and assignment

```
array[5] real y;
array[5] int x;

x = rep_array(1, 5);     // ok
y = rep_array(1.0, 5);   // ok

x = rep_array(1.0, 5);   // illegal
y = rep_array(1, 5);     // illegal

x = y;                  // illegal
y = x;                  // illegal
```

If the value being repeated `v` is a vector (i.e., `T` is `vector`), then `rep_array(v, 27)` is a size 27 array consisting of 27 copies of the vector `v`.

```
vector[5] v;
array[3] vector[5] a;

a = rep_array(v, 3);  // fill a with copies of v
a[2, 4] = 9.0;        // v[4], a[1, 4], a[3, 4] unchanged
```

If the type T of x is itself an array type, then the result will be an array with one, two, or three added dimensions, depending on which of the `rep_array` functions is called. For instance, consider the following legal code snippet.

```
array[5, 6] real a;
array[3, 4, 5, 6] real b;

b = rep_array(a, 3, 4); //  make (3 x 4) copies of a
b[1, 1, 1, 1] = 27.9;    //  a[1, 1] unchanged
```

After the assignment to `b`, the value for `b[j, k, m, n]` is equal to `a[m, n]` where it is defined, for `j` in `1:3`, `k` in `1:4`, `m` in `1:5`, and `n` in `1:6`.

5.4. Array concatenation

`T` **`append_array`**`(T x, T y)`
Return the concatenation of two arrays in the order of the arguments. T must be an N-dimensional array of any Stan type (with a maximum N of 7). All dimensions but the first must match.

Available since 2.18

For example, the following code appends two three dimensional arrays of matrices together. Note that all dimensions except the first match. Any mismatches will cause an error to be thrown.

```
array[2, 1, 7] matrix[4, 6] x1;
array[3, 1, 7] matrix[4, 6] x2;
array[5, 1, 7] matrix[4, 6] x3;

x3 = append_array(x1, x2);
```

5.5. Sorting functions

Sorting can be used to sort values or the indices of those values in either ascending or descending order. For example, if `v` is declared as a real array of size 3, with values

$$\mathrm{v} = (1, -10.3, 20.987),$$

then the various sort routines produce

$$\begin{aligned}
\mathrm{sort_asc(v)} &= (-10.3, 1, 20.987) \\
\mathrm{sort_desc(v)} &= (20.987, 1, -10.3) \\
\mathrm{sort_indices_asc(v)} &= (2, 1, 3) \\
\mathrm{sort_indices_desc(v)} &= (3, 1, 2)
\end{aligned}$$

`array[] real` **`sort_asc`**`(array[] real v)`
Sort the elements of v in ascending order

Available since 2.0

`array[] int` **`sort_asc`**`(array[] int v)`
Sort the elements of v in ascending order

Available since 2.0

`array[] real` **`sort_desc`**`(array[] real v)`
Sort the elements of v in descending order

Available since 2.0

`array[] int` **`sort_desc`**`(array[] int v)`
Sort the elements of v in descending order

Available since 2.0

`array[] int` **`sort_indices_asc`**`(array[] real v)`
Return an array of indices between 1 and the size of v, sorted to index v in ascending order.

Available since 2.3

`array[] int` **`sort_indices_asc`**`(array[] int v)`
Return an array of indices between 1 and the size of v, sorted to index v in ascending order.

Available since 2.3

`array[] int` **`sort_indices_desc`**`(array[] real v)`
Return an array of indices between 1 and the size of v, sorted to index v in descending order.

Available since 2.3

`array[] int` **`sort_indices_desc`**`(array[] int v)`
Return an array of indices between 1 and the size of v, sorted to index v in descending order.

Available since 2.3

`int` **`rank`**`(array[] real v, int s)`
Number of components of v less than v[s]

Available since 2.0

`int` **`rank`**`(array[] int v, int s)`
Number of components of v less than v[s]

Available since 2.0

5.6. Reversing functions

Stan provides functions to create a new array by reversing the order of elements in an existing array. For example, if `v` is declared as a real array of size 3, with values

$$\text{v} = (1, -10.3, 20.987),$$

then

$$\text{reverse}(\text{v}) = (20.987, -10.3, 1).$$

`array[] T` **`reverse`**`(array[] T v)`
Return a new array containing the elements of the argument in reverse order.

Available since 2.23

6. Matrix Operations

6.1. Integer-valued matrix size functions

`int` **`num_elements`**`(vector x)`
The total number of elements in the vector x (same as function `rows`)

Available since 2.5

`int` **`num_elements`**`(row_vector x)`
The total number of elements in the vector x (same as function `cols`)

Available since 2.5

`int` **`num_elements`**`(matrix x)`
The total number of elements in the matrix x. For example, if `x` is a 5×3 matrix, then `num_elements(x)` is 15

Available since 2.5

`int` **`rows`**`(vector x)`
The number of rows in the vector x

Available since 2.0

`int` **`rows`**`(row_vector x)`
The number of rows in the row vector x, namely 1

Available since 2.0

`int` **`rows`**`(matrix x)`
The number of rows in the matrix x

Available since 2.0

`int` **`cols`**`(vector x)`
The number of columns in the vector x, namely 1

Available since 2.0

`int` **`cols`**`(row_vector x)`
The number of columns in the row vector x

Available since 2.0

`int` **`cols`**`(matrix x)`
The number of columns in the matrix `x`

Available since 2.0

`int` **`size`**`(vector x)`
The size of `x`, i.e., the number of elements

Available since 2.26

`int` **`size`**`(row_vector x)`
The size of `x`, i.e., the number of elements

Available since 2.26

`int` **`size`**`(matrix x)`
The size of the matrix `x`. For example, if `x` is a 5×3 matrix, then `size(x)` is 15

Available since 2.26

6.2. Matrix arithmetic operators

Stan supports the basic matrix operations using infix, prefix and postfix operations. This section lists the operations supported by Stan along with their argument and result types.

Negation prefix operators

`vector` **`operator-`**`(vector x)`
The negation of the vector `x`.

Available since 2.0

`row_vector` **`operator-`**`(row_vector x)`
The negation of the row vector `x`.

Available since 2.0

`matrix` **`operator-`**`(matrix x)`
The negation of the matrix `x`.

Available since 2.0

`T` **`operator-`**`(T x)`
Vectorized version of `operator-`. If `T x` is a (possibly nested) array of matrix types, `-x` is the same shape array where each individual value is negated.

Available since 2.31

Infix matrix operators

`vector` **`operator+`**`(vector x, vector y)`
The sum of the vectors x and y.

Available since 2.0

`row_vector` **`operator+`**`(row_vector x, row_vector y)`
The sum of the row vectors x and y.

Available since 2.0

`matrix` **`operator+`**`(matrix x, matrix y)`
The sum of the matrices x and y

Available since 2.0

`vector` **`operator-`**`(vector x, vector y)`
The difference between the vectors x and y.

Available since 2.0

`row_vector` **`operator-`**`(row_vector x, row_vector y)`
The difference between the row vectors x and y

Available since 2.0

`matrix` **`operator-`**`(matrix x, matrix y)`
The difference between the matrices x and y

Available since 2.0

`vector` **`operator*`**`(real x, vector y)`
The product of the scalar x and vector y

Available since 2.0

`row_vector` **`operator*`**`(real x, row_vector y)`
The product of the scalar x and the row vector y

Available since 2.0

`matrix` **`operator*`**`(real x, matrix y)`
The product of the scalar x and the matrix y

Available since 2.0

`vector` **`operator*`**`(vector x, real y)`
The product of the scalar y and vector x

Available since 2.0

`matrix` **`operator*`**`(vector x, row_vector y)`
The product of the vector x and row vector y

Available since 2.0

`row_vector` **`operator*`**`(row_vector x, real y)`
The product of the scalar y and row vector x

Available since 2.0

`real` **`operator*`**`(row_vector x, vector y)`
The product of the row vector x and vector y

Available since 2.0

`row_vector` **`operator*`**`(row_vector x, matrix y)`
The product of the row vector x and matrix y

Available since 2.0

`matrix` **`operator*`**`(matrix x, real y)`
The product of the scalar y and matrix x

Available since 2.0

`vector` **`operator*`**`(matrix x, vector y)`
The product of the matrix x and vector y

Available since 2.0

`matrix` **`operator*`**`(matrix x, matrix y)`
The product of the matrices x and y

Available since 2.0

Broadcast infix operators

`vector` **`operator+`**`(vector x, real y)`
The result of adding y to every entry in the vector x

Available since 2.0

`vector` **`operator+`**`(real x, vector y)`
The result of adding x to every entry in the vector y

Available since 2.0

`row_vector` **`operator+`**`(row_vector x, real y)`
The result of adding y to every entry in the row vector x

Available since 2.0

`row_vector` **`operator+`**`(real x, row_vector y)`
The result of adding x to every entry in the row vector y

Available since 2.0

`matrix` **`operator+`**`(matrix x, real y)`
The result of adding y to every entry in the matrix x

Available since 2.0

`matrix` **`operator+`**`(real x, matrix y)`
The result of adding x to every entry in the matrix y

Available since 2.0

`vector` **`operator-`**`(vector x, real y)`
The result of subtracting y from every entry in the vector x

Available since 2.0

`vector` **`operator-`**`(real x, vector y)`
The result of adding x to every entry in the negation of the vector y

Available since 2.0

`row_vector` **`operator-`**`(row_vector x, real y)`
The result of subtracting y from every entry in the row vector x

Available since 2.0

`row_vector` **`operator-`**`(real x, row_vector y)`
The result of adding x to every entry in the negation of the row vector y

Available since 2.0

`matrix` **`operator-`**`(matrix x, real y)`
The result of subtracting y from every entry in the matrix x

Available since 2.0

`matrix` **`operator-`**`(real x, matrix y)`
The result of adding x to every entry in negation of the matrix y

Available since 2.0

`vector` **`operator/`**`(vector x, real y)`
The result of dividing each entry in the vector x by y

Available since 2.0

`row_vector` **`operator/`**`(row_vector x, real y)`
The result of dividing each entry in the row vector x by y

Available since 2.0

`matrix` **`operator/`**`(matrix x, real y)`
The result of dividing each entry in the matrix x by y

Available since 2.0

6.3. Transposition operator

Matrix transposition is represented using a postfix operator.

`matrix` **`operator'`**`(matrix x)`
The transpose of the matrix x, written as `x'`

Available since 2.0

`row_vector` **`operator'`**`(vector x)`
The transpose of the vector x, written as `x'`

Available since 2.0

`vector` **`operator'`**`(row_vector x)`
The transpose of the row vector x, written as `x'`

Available since 2.0

6.4. Elementwise functions

Elementwise functions apply a function to each element of a vector or matrix, returning a result of the same shape as the argument. There are many functions that are vectorized in addition to the ad hoc cases listed in this section; see section function vectorization for the general cases.

`vector` **`operator.*`**`(vector x, vector y)`
The elementwise product of y and x

Available since 2.0

`row_vector` **`operator.*`**`(row_vector x, row_vector y)`
The elementwise product of y and x

Available since 2.0

`matrix` **`operator.*`**`(matrix x, matrix y)`
The elementwise product of y and x

Available since 2.0

`vector` **`operator./`**`(vector x, vector y)`
The elementwise quotient of y and x

Available since 2.0

`vector` **`operator./`**`(vector x, real y)`
The elementwise quotient of y and x

Available since 2.4

`vector` **`operator./`**`(real x, vector y)`
The elementwise quotient of y and x

Available since 2.4

`row_vector` **`operator./`**`(row_vector x, row_vector y)`
The elementwise quotient of y and x

Available since 2.0

`row_vector` **`operator./`**`(row_vector x, real y)`
The elementwise quotient of y and x

Available since 2.4

`row_vector` **`operator./`**`(real x, row_vector y)`
The elementwise quotient of y and x

Available since 2.4

`matrix` **`operator./`**`(matrix x, matrix y)`
The elementwise quotient of y and x

Available since 2.0

`matrix` **`operator./`**`(matrix x, real y)`
The elementwise quotient of y and x

Available since 2.4

`matrix` **`operator./`**`(real x, matrix y)`
The elementwise quotient of y and x

Available since 2.4

`vector` **`operator.^`**`(vector x, vector y)`
The elementwise power of y and x

Available since 2.24

`vector` **`operator.^`**`(vector x, real y)`
The elementwise power of y and x

Available since 2.24

`vector` **`operator.^`**`(real x, vector y)`
The elementwise power of y and x

Available since 2.24

`row_vector` **`operator.^`**`(row_vector x, row_vector y)`
The elementwise power of y and x

Available since 2.24

`row_vector` **`operator.^`**`(row_vector x, real y)`
The elementwise power of y and x

Available since 2.24

`row_vector` **`operator.^`**`(real x, row_vector y)`
The elementwise power of y and x

Available since 2.24

`matrix` **`operator.^`**`(matrix x, matrix y)`
The elementwise power of y and x

Available since 2.24

`matrix` **`operator.^`**`(matrix x, real y)`
The elementwise power of y and x

Available since 2.24

`matrix` **`operator.^`**`(real x, matrix y)`
The elementwise power of y and x

Available since 2.24

6.5. Dot products and specialized products

`real` **`dot_product`**`(vector x, vector y)`
The dot product of x and y

Available since 2.0

`real` **`dot_product`**`(vector x, row_vector y)`
The dot product of x and y

Available since 2.0

`real` **`dot_product`**`(row_vector x, vector y)`
The dot product of x and y

Available since 2.0

`real` **`dot_product`**`(row_vector x, row_vector y)`
The dot product of x and y

Available since 2.0

`row_vector` **`columns_dot_product`**`(vector x, vector y)`
The dot product of the columns of x and y

Available since 2.0

`row_vector` **`columns_dot_product`**`(row_vector x, row_vector y)`
The dot product of the columns of x and y

Available since 2.0

`row_vector` **`columns_dot_product`**`(matrix x, matrix y)`
The dot product of the columns of x and y

Available since 2.0

`vector` **`rows_dot_product`**`(vector x, vector y)`
The dot product of the rows of x and y

Available since 2.0

`vector` **`rows_dot_product`**`(row_vector x, row_vector y)`
The dot product of the rows of x and y

Available since 2.0

`vector` **`rows_dot_product`**`(matrix x, matrix y)`
The dot product of the rows of x and y

Available since 2.0

`real` **`dot_self`**`(vector x)`
The dot product of the vector x with itself

Available since 2.0

`real` **`dot_self`**`(row_vector x)`
The dot product of the row vector x with itself

Available since 2.0

`row_vector` **`columns_dot_self`**`(vector x)`
The dot product of the columns of x with themselves

Available since 2.0

`row_vector` **`columns_dot_self`**`(row_vector x)`
The dot product of the columns of x with themselves

Available since 2.0

`row_vector` **`columns_dot_self`**`(matrix x)`
The dot product of the columns of x with themselves

Available since 2.0

`vector` **`rows_dot_self`**`(vector x)`
The dot product of the rows of x with themselves

Available since 2.0

`vector` **`rows_dot_self`**`(row_vector x)`
The dot product of the rows of x with themselves

Available since 2.0

`vector` **`rows_dot_self`**`(matrix x)`
The dot product of the rows of x with themselves

Available since 2.0

Specialized products

`matrix` **`tcrossprod`**`(matrix x)`
The product of x postmultiplied by its own transpose, similar to the tcrossprod(x) function in R. The result is a symmetric matrix $\mathrm{x}\,\mathrm{x}^\top$.

Available since 2.0

`matrix` **`crossprod`**`(matrix x)`
The product of x premultiplied by its own transpose, similar to the crossprod(x) function in R. The result is a symmetric matrix $x^\top x$.

Available since 2.0

The following functions all provide shorthand forms for common expressions, which are also much more efficient.

`matrix` **`quad_form`**`(matrix A, matrix B)`
The quadratic form, i.e., `B' * A * B`.

Available since 2.0

`real` **`quad_form`**`(matrix A, vector B)`
The quadratic form, i.e., `B' * A * B`.

Available since 2.0

`matrix` **`quad_form_diag`**`(matrix m, vector v)`
The quadratic form using the column vector v as a diagonal matrix, i.e., `diag_matrix(v) * m * diag_matrix(v)`.

Available since 2.3

`matrix` **`quad_form_diag`**`(matrix m, row_vector rv)`
The quadratic form using the row vector rv as a diagonal matrix, i.e., `diag_matrix(rv) * m * diag_matrix(rv)`.

Available since 2.3

`matrix` **`quad_form_sym`**`(matrix A, matrix B)`
Similarly to quad_form, gives `B' * A * B`, but additionally checks if A is symmetric and ensures that the result is also symmetric.

Available since 2.3

`real` **`quad_form_sym`**`(matrix A, vector B)`
Similarly to quad_form, gives `B' * A * B`, but additionally checks if A is symmetric and ensures that the result is also symmetric.

Available since 2.3

`real` **`trace_quad_form`**`(matrix A, matrix B)`
The trace of the quadratic form, i.e., `trace(B' * A * B)`.

Available since 2.0

`real` **`trace_gen_quad_form`**`(matrix D,matrix A, matrix B)`
The trace of a generalized quadratic form, i.e., `trace(D * B' * A * B).`

Available since 2.0

`matrix` **`multiply_lower_tri_self_transpose`**`(matrix x)`
The product of the lower triangular portion of `x` (including the diagonal) times its own transpose; that is, if `L` is a matrix of the same dimensions as `x` with `L(m,n)` equal to `x(m,n)` for $\mathrm{n} \leq \mathrm{m}$ and `L(m,n)` equal to 0 if $\mathrm{n} > \mathrm{m}$, the result is the symmetric matrix $\mathrm{L}\,\mathrm{L}^\top$. This is a specialization of tcrossprod(x) for lower-triangular matrices. The input matrix does not need to be square.

Available since 2.0

`matrix` **`diag_pre_multiply`**`(vector v, matrix m)`
Return the product of the diagonal matrix formed from the vector v and the matrix m, i.e., `diag_matrix(v) * m`.

Available since 2.0

`matrix` **`diag_pre_multiply`**`(row_vector rv, matrix m)`
Return the product of the diagonal matrix formed from the vector rv and the matrix m, i.e., `diag_matrix(rv) * m`.

Available since 2.0

`matrix` **`diag_post_multiply`**`(matrix m, vector v)`
Return the product of the matrix m and the diagonal matrix formed from the vector v, i.e., `m * diag_matrix(v)`.

Available since 2.0

`matrix` **`diag_post_multiply`**`(matrix m, row_vector rv)`
Return the product of the matrix `m` and the diagonal matrix formed from the the row vector `rv`, i.e., `m * diag_matrix(rv)`.

Available since 2.0

6.6. Reductions

Log sum of exponents

`real` **`log_sum_exp`**`(vector x)`
The natural logarithm of the sum of the exponentials of the elements in x

Available since 2.0

`real` **`log_sum_exp`**`(row_vector x)`
The natural logarithm of the sum of the exponentials of the elements in x

Available since 2.0

`real` **`log_sum_exp`**`(matrix x)`
The natural logarithm of the sum of the exponentials of the elements in x

Available since 2.0

Minimum and maximum

`real` **`min`**`(vector x)`
The minimum value in x, or $+\infty$ if x is empty

Available since 2.0

`real` **`min`**`(row_vector x)`
The minimum value in x, or $+\infty$ if x is empty

Available since 2.0

`real` **`min`**`(matrix x)`
The minimum value in x, or $+\infty$ if x is empty

Available since 2.0

`real` **`max`**`(vector x)`
The maximum value in x, or $-\infty$ if x is empty

Available since 2.0

`real` **`max`**`(row_vector x)`
The maximum value in x, or $-\infty$ if x is empty

Available since 2.0

`real` **`max`**`(matrix x)`
The maximum value in x, or $-\infty$ if x is empty

Available since 2.0

Sums and products

`real` **`sum`**`(vector x)`
The sum of the values in x, or 0 if x is empty

Available since 2.0

`real` **`sum`**`(row_vector x)`
The sum of the values in x, or 0 if x is empty

Available since 2.0

`real` **`sum`**`(matrix x)`
The sum of the values in x, or 0 if x is empty

Available since 2.0

`real` **`prod`**`(vector x)`
The product of the values in x, or 1 if x is empty

Available since 2.0

`real` **`prod`**`(row_vector x)`
The product of the values in x, or 1 if x is empty

Available since 2.0

`real` **`prod`**`(matrix x)`
The product of the values in x, or 1 if x is empty

Available since 2.0

Sample moments

Full definitions are provided for sample moments in section array reductions.

`real` **`mean`**`(vector x)`
The sample mean of the values in x; see section array reductions for details.

Available since 2.0

`real` **`mean`**`(row_vector x)`
The sample mean of the values in x; see section array reductions for details.

Available since 2.0

`real` **`mean`**`(matrix x)`
The sample mean of the values in x; see section array reductions for details.

Available since 2.0

`real` **`variance`**`(vector x)`
The sample variance of the values in x; see section array reductions for details.

Available since 2.0

`real` **`variance`**`(row_vector x)`
The sample variance of the values in x; see section array reductions for details.

Available since 2.0

`real` **`variance`**`(matrix x)`
The sample variance of the values in x; see section array reductions for details.

Available since 2.0

`real` **`sd`**`(vector x)`
The sample standard deviation of the values in x; see section array reductions for details.

Available since 2.0

`real` **`sd`**`(row_vector x)`
The sample standard deviation of the values in x; see section array reductions for details.

Available since 2.0

`real` **`sd`**`(matrix x)`
The sample standard deviation of the values in x; see section array reductions for details.

Available since 2.0

Quantile

Produces sample quantiles corresponding to the given probabilities. The smallest observation corresponds to a probability of 0 and the largest to a probability of 1.

Implements algorithm 7 from Hyndman, R. J. and Fan, Y., Sample quantiles in Statistical Packages (R's default quantile function).

`real` **`quantile`**`(data vector x, data real p)`
The p-th quantile of x

Available since 2.27

`array[] real` **`quantile`**`(data vector x, data array[] real p)`
An array containing the quantiles of x given by the array of probabilities p

Available since 2.27

`real` **`quantile`**`(data row_vector x, data real p)`
The p-th quantile of x

Available since 2.27

`array[] real` **`quantile`**`(data row_vector x, data array[] real p)`
An array containing the quantiles of x given by the array of probabilities p

Available since 2.27

6.7. Broadcast functions

The following broadcast functions allow vectors, row vectors and matrices to be created by copying a single element into all of their cells. Matrices may also be created by stacking copies of row vectors vertically or stacking copies of column vectors horizontally.

`vector` **`rep_vector`**`(real x, int m)`
Return the size m (column) vector consisting of copies of x.

Available since 2.0

`row_vector` **`rep_row_vector`**`(real x, int n)`
Return the size n row vector consisting of copies of x.

Available since 2.0

`matrix` **`rep_matrix`**`(real x, int m, int n)`
Return the m by n matrix consisting of copies of x.

Available since 2.0

`matrix` **`rep_matrix`**`(vector v, int n)`
Return the m by n matrix consisting of n copies of the (column) vector v of size m.

Available since 2.0

`matrix` **`rep_matrix`**`(row_vector rv, int m)`
Return the m by n matrix consisting of m copies of the row vector rv of size n.

Available since 2.0

Unlike the situation with array broadcasting (see section array broadcasting), where there is a distinction between integer and real arguments, the following two statements produce the same result for vector broadcasting; row vector and matrix broadcasting behave similarly.

```
vector[3] x;
x = rep_vector(1, 3);
x = rep_vector(1.0, 3);
```

There are no integer vector or matrix types, so integer values are automatically promoted.

Symmetrization

`matrix` **`symmetrize_from_lower_tri`**`(matrix A)`

Construct a symmetric matrix from the lower triangle of A.

Available since 2.26

6.8. Diagonal matrix functions

`matrix` **`add_diag`**`(matrix m, row_vector d)`
Add row_vector `d` to the diagonal of matrix `m`.

Available since 2.21

`matrix` **`add_diag`**`(matrix m, vector d)`
Add vector `d` to the diagonal of matrix `m`.

Available since 2.21

`matrix` **`add_diag`**`(matrix m, real d)`
Add scalar `d` to every diagonal element of matrix `m`.

Available since 2.21

`vector` **`diagonal`**`(matrix x)`
The diagonal of the matrix x

Available since 2.0

`matrix` **`diag_matrix`**`(vector x)`
The diagonal matrix with diagonal x

Available since 2.0

Although the `diag_matrix` function is available, it is unlikely to ever show up in an efficient Stan program. For example, rather than converting a diagonal to a full matrix for use as a covariance matrix,

```
y ~ multi_normal(mu, diag_matrix(square(sigma)));
```

it is much more efficient to just use a univariate normal, which produces the same density,

```
y ~ normal(mu, sigma);
```

Rather than writing `m * diag_matrix(v)` where `m` is a matrix and `v` is a vector, it is much more efficient to write `diag_post_multiply(m, v)` (and similarly for pre-multiplication). By the same token, it is better to use `quad_form_diag(m, v)` rather than `quad_form(m, diag_matrix(v))`.

`matrix` **`identity_matrix`**`(int k)`
Create an identity matrix of size $k \times k$

Available since 2.26

6.9. Container construction functions

`array[] real` **`linspaced_array`**`(int n, data real lower, data real upper)`
Create a real array of length `n` of equidistantly-spaced elements between `lower` and `upper`

Available since 2.24

`array[] int` **`linspaced_int_array`**`(int n, int lower, int upper)`
Create a regularly spaced, increasing integer array of length `n` between `lower` and `upper`, inclusively. If `(upper - lower) / (n - 1)` is less than one, repeat each output `(n - 1) / (upper - lower)` times. If neither `(upper - lower) / (n - 1)` or `(n - 1) / (upper - lower)` are integers, `upper` is reduced until one of these is true.

Available since 2.26

`vector` **`linspaced_vector`**`(int n, data real lower, data real upper)`
Create an `n`-dimensional vector of equidistantly-spaced elements between `lower` and `upper`

Available since 2.24

`row_vector` **`linspaced_row_vector`**`(int n, data real lower, data real upper)`
Create an `n`-dimensional row-vector of equidistantly-spaced elements between `lower` and `upper`

Available since 2.24

`array[] int` **`one_hot_int_array`**`(int n, int k)`
Create a one-hot encoded int array of length `n` with `array[k] = 1`

Available since 2.26

`array[] real` **`one_hot_array`**`(int n, int k)`
Create a one-hot encoded real array of length `n` with `array[k] = 1`

Available since 2.24

`vector` **`one_hot_vector`**`(int n, int k)`
Create an `n`-dimensional one-hot encoded vector with `vector[k] = 1`

Available since 2.24

`row_vector` **`one_hot_row_vector`**`(int n, int k)`
Create an `n`-dimensional one-hot encoded row-vector with `row_vector[k] = 1`

Available since 2.24

`array[] int` **`ones_int_array`**`(int n)`
Create an int array of length `n` of all ones

Available since 2.26

`array[] real` **`ones_array`**`(int n)`
Create a real array of length `n` of all ones

Available since 2.26

`vector` **`ones_vector`**`(int n)`
Create an `n`-dimensional vector of all ones

Available since 2.26

`row_vector` **`ones_row_vector`**`(int n)`
Create an `n`-dimensional row-vector of all ones

Available since 2.26

`array[] int` **`zeros_int_array`**`(int n)`
Create an int array of length `n` of all zeros

Available since 2.26

`array[] real` **`zeros_array`**`(int n)`
Create a real array of length `n` of all zeros

Available since 2.24

`vector` **`zeros_vector`**`(int n)`
Create an `n`-dimensional vector of all zeros

Available since 2.24

`row_vector` **`zeros_row_vector`**`(int n)`
Create an `n`-dimensional row-vector of all zeros

Available since 2.24

`vector` **`uniform_simplex`**`(int n)`
Create an `n`-dimensional simplex with elements `vector[i] = 1 / n` for all $i \in 1, \ldots, n$

Available since 2.24

6.10. Slicing and blocking functions

Stan provides several functions for generating slices or blocks or diagonal entries for matrices.

Columns and rows

`vector` **`col`**`(matrix x, int n)`
The n-th column of matrix x

Available since 2.0

`row_vector` **`row`**`(matrix x, int m)`
The m-th row of matrix x

Available since 2.0

The `row` function is special in that it may be used as an lvalue in an assignment statement (i.e., something to which a value may be assigned). The row function is also special in that the indexing notation `x[m]` is just an alternative way of writing `row(x,m)`. The `col` function may **not**, be used as an lvalue, nor is there an indexing based shorthand for it.

Block operations

Matrix slicing operations

Block operations may be used to extract a sub-block of a matrix.

`matrix` **`block`**`(matrix x, int i, int j, int n_rows, int n_cols)`
Return the submatrix of x that starts at row i and column j and extends n_rows rows and n_cols columns.

Available since 2.0

The sub-row and sub-column operations may be used to extract a slice of row or column from a matrix

`vector` **`sub_col`**`(matrix x, int i, int j, int n_rows)`
Return the sub-column of x that starts at row i and column j and extends n_rows rows and 1 column.

Available since 2.0

`row_vector` **`sub_row`**`(matrix x, int i, int j, int n_cols)`
Return the sub-row of x that starts at row i and column j and extends 1 row and n_cols columns.

Available since 2.0

Vector and array slicing operations

The head operation extracts the first n elements of a vector and the tail operation the last. The segment operation extracts an arbitrary subvector.

`vector` **`head`**`(vector v, int n)`
Return the vector consisting of the first n elements of v.

Available since 2.0

`row_vector` **`head`**`(row_vector rv, int n)`
Return the row vector consisting of the first n elements of rv.

Available since 2.0

`array[] T` **`head`**`(array[] T sv, int n)`
Return the array consisting of the first n elements of sv; applies to up to three-dimensional arrays containing any type of elements `T`.

Available since 2.0

`vector` **`tail`**`(vector v, int n)`
Return the vector consisting of the last n elements of v.

Available since 2.0

`row_vector` **`tail`**`(row_vector rv, int n)`
Return the row vector consisting of the last n elements of rv.

Available since 2.0

`array[] T` **`tail`**`(array[] T sv, int n)`

Return the array consisting of the last n elements of sv; applies to up to three-dimensional arrays containing any type of elements `T`.

Available since 2.0

`vector` **`segment`**`(vector v, int i, int n)`
Return the vector consisting of the n elements of v starting at i; i.e., elements i through through i + n - 1.

Available since 2.0

`row_vector` **`segment`**`(row_vector rv, int i, int n)`
Return the row vector consisting of the n elements of rv starting at i; i.e., elements i through through i + n - 1.

Available since 2.10

`array[] T` **`segment`**`(array[] T sv, int i, int n)`
Return the array consisting of the n elements of sv starting at i; i.e., elements i through through i + n - 1. Applies to up to three-dimensional arrays containing any type of elements `T`.

Available since 2.0

6.11. Matrix concatenation

Stan's matrix concatenation operations `append_col` and `append_row` are like the operations `cbind` and `rbind` in R.

Horizontal concatenation

`matrix` **`append_col`**`(matrix x, matrix y)`
Combine matrices x and y by column. The matrices must have the same number of rows.

Available since 2.5

`matrix` **`append_col`**`(matrix x, vector y)`
Combine matrix x and vector y by column. The matrix and the vector must have the same number of rows.

Available since 2.5

`matrix` **`append_col`**`(vector x, matrix y)`
Combine vector x and matrix y by column. The vector and the matrix must have the same number of rows.

Available since 2.5

`matrix` **`append_col`**`(vector x, vector y)`
Combine vectors x and y by column. The vectors must have the same number of rows.

Available since 2.5

`row_vector` **`append_col`**`(row_vector x, row_vector y)`
Combine row vectors x and y of any size into another row vector by appending y to the end of x.

Available since 2.5

`row_vector` **`append_col`**`(real x, row_vector y)`
Append x to the front of y, returning another row vector.

Available since 2.12

`row_vector` **`append_col`**`(row_vector x, real y)`
Append y to the end of x, returning another row vector.

Available since 2.12

Vertical concatenation

`matrix` **`append_row`**`(matrix x, matrix y)`
Combine matrices x and y by row. The matrices must have the same number of columns.

Available since 2.5

`matrix` **`append_row`**`(matrix x, row_vector y)`
Combine matrix x and row vector y by row. The matrix and the row vector must have the same number of columns.

Available since 2.5

`matrix` **`append_row`**`(row_vector x, matrix y)`
Combine row vector x and matrix y by row. The row vector and the matrix must have the same number of columns.

Available since 2.5

`matrix` **`append_row`**`(row_vector x, row_vector y)`
Combine row vectors x and y by row. The row vectors must have the same number of columns.

Available since 2.5

`vector` **`append_row`**`(vector x, vector y)`
Concatenate vectors x and y of any size into another vector.

Available since 2.5

`vector` **`append_row`**`(real x, vector y)`
Append x to the top of y, returning another vector.

Available since 2.12

`vector` **`append_row`**`(vector x, real y)`
Append y to the bottom of x, returning another vector.

Available since 2.12

6.12. Special matrix functions

Softmax

The softmax function maps[1] $y \in \mathbb{R}^K$ to the K-simplex by

$$\text{softmax}(y) = \frac{\exp(y)}{\sum_{k=1}^{K} \exp(y_k)},$$

where $\exp(y)$ is the componentwise exponentiation of y. Softmax is usually calculated on the log scale,

$$\begin{aligned} \log \text{softmax}(y) &= y - \log \sum_{k=1}^{K} \exp(y_k) \\ &= y - \text{log\_sum\_exp}(y). \end{aligned}$$

where the vector y minus the scalar $\text{log\_sum\_exp}(y)$ subtracts the scalar from each component of y.

Stan provides the following functions for softmax and its log.

`vector` **`softmax`**`(vector x)`
The softmax of x

Available since 2.0

`vector` **`log_softmax`**`(vector x)`
The natural logarithm of the softmax of x

[1]The softmax function is so called because in the limit as $y_n \to \infty$ with y_m for $m \neq n$ held constant, the result tends toward the "one-hot" vector θ with $\theta_n = 1$ and $\theta_m = 0$ for $m \neq n$, thus providing a "soft" version of the maximum function.

Available since 2.0

Cumulative sums

The cumulative sum of a sequence $x_1, \ldots, x_N$ is the sequence $y_1, \ldots, y_N$, where

$$y_n = \sum_{m=1}^{n} x_m.$$

`array[] int` **`cumulative_sum`**`(array[] int x)`
The cumulative sum of x

Available since 2.30

`array[] real` **`cumulative_sum`**`(array[] real x)`
The cumulative sum of x

Available since 2.0

`vector` **`cumulative_sum`**`(vector v)`
The cumulative sum of v

Available since 2.0

`row_vector` **`cumulative_sum`**`(row_vector rv)`
The cumulative sum of rv

Available since 2.0

6.13. Gaussian Process Covariance Functions

The Gaussian process covariance functions compute the covariance between observations in an input data set or the cross-covariance between two input data sets.

For one dimensional GPs, the input data sets are arrays of scalars. The covariance matrix is given by $K_{ij} = k(x_i, x_j)$ (where x_i is the i^{th} element of the array x) and the cross-covariance is given by $K_{ij} = k(x_i, y_j)$.

For multi-dimensional GPs, the input data sets are arrays of vectors. The covariance matrix is given by $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$ (where $\mathbf{x}_i$ is the i^{th} vector in the array x) and the cross-covariance is given by $K_{ij} = k(\mathbf{x}_i, \mathbf{y}_j)$.

Exponentiated quadratic kernel

With magnitude σ and length scale l, the exponentiated quadratic kernel is:

$$k(\mathbf{x}_i, \mathbf{x}_j) = \sigma^2 \exp\left(-\frac{|\mathbf{x}_i - \mathbf{x}_j|^2}{2l^2}\right)$$

`matrix` **`gp_exp_quad_cov`**`(array[] real x, real sigma, real length_scale)`

Gaussian process covariance with exponentiated quadratic kernel in one dimension.

Available since 2.20

`matrix` **`gp_exp_quad_cov`**`(array[] real x1, array[] real x2, real sigma, real length_scale)`

Gaussian process cross-covariance of `x1` and `x2` with exponentiated quadratic kernel in one dimension.

Available since 2.20

`matrix` **`gp_exp_quad_cov`**`(vectors x, real sigma, real length_scale)`

Gaussian process covariance with exponentiated quadratic kernel in multiple dimensions.

Available since 2.20

`matrix` **`gp_exp_quad_cov`**`(vectors x, real sigma, array[] real length_scale)`

Gaussian process covariance with exponentiated quadratic kernel in multiple dimensions with a length scale for each dimension.

Available since 2.20

`matrix` **`gp_exp_quad_cov`**`(vectors x1, vectors x2, real sigma, real length_scale)`

Gaussian process cross-covariance of `x1` and `x2` with exponentiated quadratic kernel in multiple dimensions.

Available since 2.20

`matrix` **`gp_exp_quad_cov`**`(vectors x1, vectors x2, real sigma, array[] real length_scale)`

Gaussian process cross-covariance of `x1` and `x2` with exponentiated quadratic kernel in multiple dimensions with a length scale for each dimension.

Available since 2.20

Dot product kernel

With bias σ_0 the dot product kernel is:

$$k(\mathbf{x}_i, \mathbf{x}_j) = \sigma_0^2 + \mathbf{x}_i^T \mathbf{x}_j$$

`matrix` **`gp_dot_prod_cov`**`(array[] real x, real sigma)`

Gaussian process covariance with dot product kernel in one dimension.

Available since 2.20

`matrix` **`gp_dot_prod_cov`**`(array[] real x1, array[] real x2, real sigma)`

Gaussian process cross-covariance of `x1` and `x2` with dot product kernel in one dimension.

Available since 2.20

`matrix` **`gp_dot_prod_cov`**`(vectors x, real sigma)`

Gaussian process covariance with dot product kernel in multiple dimensions.

Available since 2.20

`matrix` **`gp_dot_prod_cov`**`(vectors x1, vectors x2, real sigma)`

Gaussian process cross-covariance of `x1` and `x2` with dot product kernel in multiple dimensions.

Available since 2.20

Exponential kernel

With magnitude σ and length scale l, the exponential kernel is:

$$k(\mathbf{x}_i, \mathbf{x}_j) = \sigma^2 \exp\left(-\frac{|\mathbf{x}_i - \mathbf{x}_j|}{l}\right)$$

`matrix` **`gp_exponential_cov`**`(array[] real x, real sigma, real length_scale)`

Gaussian process covariance with exponential kernel in one dimension.

Available since 2.20

`matrix` **`gp_exponential_cov`**`(array[] real x1, array[] real x2, real sigma, real length_scale)`

Gaussian process cross-covariance of `x1` and `x2` with exponential kernel in one dimension.

Available since 2.20

`matrix` **`gp_exponential_cov`**`(vectors x, real sigma, real length_scale)`

Gaussian process covariance with exponential kernel in multiple dimensions.

Available since 2.20

`matrix` **`gp_exponential_cov`**`(vectors x, real sigma, array[] real length_scale)`

Gaussian process covariance with exponential kernel in multiple dimensions with a length scale for each dimension.

Available since 2.20

`matrix` **`gp_exponential_cov`**`(vectors x1, vectors x2, real sigma, real length_scale)`

Gaussian process cross-covariance of `x1` and `x2` with exponential kernel in multiple dimensions.

Available since 2.20

`matrix` **`gp_exponential_cov`**`(vectors x1, vectors x2, real sigma, array[] real length_scale)`

Gaussian process cross-covariance of `x1` and `x2` with exponential kernel in multiple dimensions with a length scale for each dimension.

Available since 2.20

Matern 3/2 kernel

With magnitude σ and length scale l, the Matern 3/2 kernel is:

$$k(\mathbf{x}_i, \mathbf{x}_j) = \sigma^2 \left(1 + \frac{\sqrt{3}|\mathbf{x}_i - \mathbf{x}_j|}{l}\right) \exp\left(-\frac{\sqrt{3}|\mathbf{x}_i - \mathbf{x}_j|}{l}\right)$$

`matrix` **`gp_matern32_cov`**`(array[] real x, real sigma, real length_scale)`

Gaussian process covariance with Matern 3/2 kernel in one dimension.

Available since 2.20

`matrix` **`gp_matern32_cov`**`(array[] real x1, array[] real x2, real sigma, real length_scale)`

Gaussian process cross-covariance of `x1` and `x2` with Matern 3/2 kernel in one dimension.

Available since 2.20

`matrix` **`gp_matern32_cov`**`(vectors x, real sigma, real length_scale)`

Gaussian process covariance with Matern 3/2 kernel in multiple dimensions.

Available since 2.20

`matrix` **`gp_matern32_cov`**`(vectors x, real sigma, array[] real length_scale)`

Gaussian process covariance with Matern 3/2 kernel in multiple dimensions with a length scale for each dimension.

Available since 2.20

`matrix` **`gp_matern32_cov`**`(vectors x1, vectors x2, real sigma, real length_scale)`

Gaussian process cross-covariance of `x1` and `x2` with Matern 3/2 kernel in multiple dimensions.

Available since 2.20

`matrix` **`gp_matern32_cov`**`(vectors x1, vectors x2, real sigma, array[] real length_scale)`

Gaussian process cross-covariance of `x1` and `x2` with Matern 3/2 kernel in multiple dimensions with a length scale for each dimension.

Available since 2.20

Matern 5/2 kernel

With magnitude σ and length scale l, the Matern 5/2 kernel is:

$$k(\mathbf{x}_i, \mathbf{x}_j) = \sigma^2 \left(1 + \frac{\sqrt{5}|\mathbf{x}_i - \mathbf{x}_j|}{l} + \frac{5|\mathbf{x}_i - \mathbf{x}_j|^2}{3l^2} \right) \exp\left(-\frac{\sqrt{5}|\mathbf{x}_i - \mathbf{x}_j|}{l} \right)$$

`matrix` **`gp_matern52_cov`**`(array[] real x, real sigma, real length_scale)`

Gaussian process covariance with Matern 5/2 kernel in one dimension.

Available since 2.20

`matrix` **`gp_matern52_cov`**`(array[] real x1, array[] real x2, real sigma, real length_scale)`

Gaussian process cross-covariance of `x1` and `x2` with Matern 5/2 kernel in one dimension.

Available since 2.20

`matrix` **`gp_matern52_cov`**`(vectors x, real sigma, real length_scale)`

Gaussian process covariance with Matern 5/2 kernel in multiple dimensions.

Available since 2.20

`matrix` **`gp_matern52_cov`**`(vectors x, real sigma, array[] real length_scale)`

Gaussian process covariance with Matern 5/2 kernel in multiple dimensions with a length scale for each dimension.

Available since 2.20

`matrix` **`gp_matern52_cov`**`(vectors x1, vectors x2, real sigma, real length_scale)`

Gaussian process cross-covariance of `x1` and `x2` with Matern 5/2 kernel in multiple dimensions.

Available since 2.20

`matrix` **`gp_matern52_cov`**`(vectors x1, vectors x2, real sigma, array[] real length_scale)`

Gaussian process cross-covariance of `x1` and `x2` with Matern 5/2 kernel in multiple dimensions with a length scale for each dimension.

Available since 2.20

Periodic kernel

With magnitude σ, length scale l, and period p, the periodic kernel is:

$$k(\mathbf{x}_i, \mathbf{x}_j) = \sigma^2 \exp\left(-\frac{2\sin^2\left(\pi \frac{|\mathbf{x}_i - \mathbf{x}_j|}{p}\right)}{l^2}\right)$$

`matrix` **`gp_periodic_cov`**`(array[] real x, real sigma, real length_scale, real period)`

Gaussian process covariance with periodic kernel in one dimension.

Available since 2.20

`matrix` **`gp_periodic_cov`**`(array[] real x1, array[] real x2, real sigma, real length_scale, real period)`

Gaussian process cross-covariance of `x1` and `x2` with periodic kernel in one dimension.

Available since 2.20

`matrix` **`gp_periodic_cov`**`(vectors x, real sigma, real length_scale, real period)`

Gaussian process covariance with periodic kernel in multiple dimensions.

Available since 2.20

`matrix` **`gp_periodic_cov`**`(vectors x1, vectors x2, real sigma, real length_scale, real period)`

Gaussian process cross-covariance of `x1` and `x2` with periodic kernel in multiple dimensions with a length scale for each dimension.

Available since 2.20

6.14. Linear algebra functions and solvers

Matrix division operators and functions

In general, it is much more efficient and also more arithmetically stable to use matrix division than to multiply by an inverse. There are specialized forms for lower triangular matrices and for symmetric, positive-definite matrices.

Matrix division operators

`row_vector` **`operator/`**`(row_vector b, matrix A)`
The right division of b by A; equivalently `b * inverse(A)`

Available since 2.0

`matrix` **`operator/`**`(matrix B, matrix A)`
The right division of B by A; equivalently `B * inverse(A)`

Available since 2.5

`vector` **`operator\`**`(matrix A, vector b)`
The left division of A by b; equivalently `inverse(A) * b`

Available since 2.18

`matrix` **`operator\`**`(matrix A, matrix B)`
The left division of A by B; equivalently `inverse(A) * B`

Available since 2.18

Lower-triangular matrix division functions

There are four division functions which use lower triangular views of a matrix. The lower triangular view of a matrix tri(A) is used in the definitions and defined by

$$\text{tri}(A)[m,n] = \begin{cases} A[m,n] & \text{if } m \geq n, \text{ and} \\ 0 & \text{otherwise.} \end{cases}$$

When a lower triangular view of a matrix is used, the elements above the diagonal are ignored.

`vector` **`mdivide_left_tri_low`**`(matrix A, vector b)`
The left division of `b` by a lower-triangular view of A; algebraically equivalent to the less efficient and stable form `inverse(tri(A)) * b`, where `tri(A)` is the lower-triangular portion of A with the above-diagonal entries set to zero.

Available since 2.12

`matrix` **`mdivide_left_tri_low`**`(matrix A, matrix B)`
The left division of B by a triangular view of A; algebraically equivalent to the less efficient and stable form `inverse(tri(A)) * B`, where `tri(A)` is the lower-triangular portion of A with the above-diagonal entries set to zero.

Available since 2.5

`row_vector` **`mdivide_right_tri_low`**`(row_vector b, matrix A)`
The right division of b by a triangular view of A; algebraically equivalent to the less efficient and stable form `b * inverse(tri(A))`, where `tri(A)` is the lower-triangular portion of A with the above-diagonal entries set to zero.

Available since 2.12

`matrix` **`mdivide_right_tri_low`**`(matrix B, matrix A)`
The right division of B by a triangular view of A; algebraically equivalent to the less efficient and stable form `B * inverse(tri(A))`, where `tri(A)` is the lower-triangular portion of A with the above-diagonal entries set to zero.

Available since 2.5

Symmetric positive-definite matrix division functions

There are four division functions which are specialized for efficiency and stability for symmetric positive-definite matrix dividends. If the matrix dividend argument is not symmetric and positive definite, these will reject and print warnings.

`matrix` **`mdivide_left_spd`**`(matrix A, vector b)`
The left division of b by the symmetric, positive-definite matrix A; algebraically equivalent to the less efficient and stable form `inverse(A) * b`.

Available since 2.12

`vector` **`mdivide_left_spd`**`(matrix A, matrix B)`
The left division of B by the symmetric, positive-definite matrix A; algebraically equivalent to the less efficient and stable form `inverse(A) * B`.

Available since 2.12

`row_vector` **`mdivide_right_spd`**`(row_vector b, matrix A)`
The right division of b by the symmetric, positive-definite matrix A; algebraically equivalent to the less efficient and stable form `b *inverse(A)`.

Available since 2.12

`matrix` **`mdivide_right_spd`**`(matrix B, matrix A)`
The right division of B by the symmetric, positive-definite matrix A; algebraically equivalent to the less efficient and stable form `B * inverse(A)`.

Available since 2.12

Matrix exponential

The exponential of the matrix A is formally defined by the convergent power series:

$$e^A = \sum_{n=0}^{\infty} \frac{A^n}{n!}$$

`matrix` **`matrix_exp`**`(matrix A)`
The matrix exponential of A

Available since 2.13

`matrix` **`matrix_exp_multiply`**`(matrix A, matrix B)`
The multiplication of matrix exponential of A and matrix B; algebraically equivalent to the less efficient form `matrix_exp(A) * B`.

Available since 2.18

`matrix` **`scale_matrix_exp_multiply`**`(real t, matrix A, matrix B)`
The multiplication of matrix exponential of tA and matrix B; algebraically equivalent to the less efficient form `matrix_exp(t * A) * B`.

Available since 2.18

Matrix power

Returns the nth power of the specific matrix:

$$M^n = M_1 * ... * M_n$$

`matrix` **`matrix_power`**`(matrix A, int B)`
Matrix A raised to the power B.

Available since 2.24

Linear algebra functions

Trace

`real` **`trace`**`(matrix A)`
The trace of A, or 0 if A is empty; A is not required to be diagonal

Available since 2.0

Determinants

`real` **`determinant`**`(matrix A)`
The determinant of A

Available since 2.0

`real` **`log_determinant`**`(matrix A)`
The log of the absolute value of the determinant of A

Available since 2.0

`real` **`log_determinant_spd`**`(matrix A)`
The log of the absolute value of the determinant of the symmetric, positive-definite matrix A.

Available since 2.30

Inverses

It is almost never a good idea to use matrix inverses directly because they are both inefficient and arithmetically unstable compared to the alternatives. Rather than inverting a matrix `m` and post-multiplying by a vector or matrix `a`, as in `inverse(m) * a`, it is better to code this using matrix division, as in `m \ a`. The pre-multiplication case is similar, with `b * inverse(m)` being more efficiently coded as as `b / m`. There are also useful special cases for triangular and symmetric, positive-definite matrices that use more efficient solvers.

Warning: The function `inv(m)` is the elementwise inverse function, which returns `1 / m[i, j]` for each element.

`matrix` **`inverse`**`(matrix A)`
Compute the inverse of A

Available since 2.0

`matrix` **`inverse_spd`**`(matrix A)`
Compute the inverse of A where A is symmetric, positive definite. This version is faster and more arithmetically stable when the input is symmetric and positive definite.

Available since 2.0

`matrix` **`chol2inv`**`(matrix L)`
Compute the inverse of the matrix whose cholesky factorization is L. That is, for $A = LL^T$, return A^{-1}.

Available since 2.26

Generalized Inverse

The generalized inverse M^+ of a matrix M is a matrix that satisfies $MM^+M = M$. For an invertible, square matrix M, M^+ is equivalent to M^{-1}. The dimensions of M^+ are equivalent to the dimensions of M^T. The generalized inverse exists for any matrix, so the M may be singular or less than full rank.

Even though the generalized inverse exists for any arbitrary matrix, the derivatives of this function only exist on matrices of locally constant rank (Golub and Pereyra 1973), meaning, the derivatives do not exist if small perturbations make the matrix change rank. For example, considered the rank of the matrix A as a function of ϵ:

$$A = \begin{pmatrix} 1+\epsilon & 2 & 1 \\ 2 & 4 & 2 \end{pmatrix}$$

When $\epsilon = 0$, A is rank 1 because the second row is twice the first (and so there is only one linearly independent row). If $\epsilon \neq 0$, the rows are no longer linearly dependent, and the matrix is rank 2. This matrix does not have locally constant rank at $\epsilon = 0$, and so the derivatives do not exist at zero. Because HMC depends on the derivatives existing, this lack of differentiability creates undefined behavior.

`matrix` **`generalized_inverse`**`(matrix A)`
The generalized inverse of A

Available since 2.26

Eigendecomposition

`complex_vector` **`eigenvalues`**`(matrix A)`
The complex-valued vector of eigenvalues of the matrix `A`. The eigenvalues are repeated according to their algebraic multiplicity, so there are as many eigenvalues as rows in the matrix. The eigenvalues are not sorted in any particular order.

Available since 2.30

`complex_matrix` **`eigenvectors`**`(matrix A)`
The matrix with the complex-valued (column) eigenvectors of the matrix `A` in the same order as returned by the function `eigenvalues`

Available since 2.30

`tuple(complex_matrix, complex_vector)` **`eigendecompose`**`(matrix A)`
Return the matrix of (column) eigenvectors and vector of eigenvalues of the matrix `A`. This function is equivalent to `(eigenvectors(A), eigenvalues(A))` but with a lower computational cost due to the shared work between the two results.

Available since 2.33

`vector` **`eigenvalues_sym`**`(matrix A)`
The vector of eigenvalues of a symmetric matrix `A` in ascending order

Available since 2.0

`matrix` **`eigenvectors_sym`**`(matrix A)`
The matrix with the (column) eigenvectors of symmetric matrix `A` in the same order as returned by the function `eigenvalues_sym`

Available since 2.0

`tuple(matrix, vector)` **`eigendecompose_sym`**`(matrix A)`
Return the matrix of (column) eigenvectors and vector of eigenvalues of the symmetric matrix `A`. This function is equivalent to `(eigenvectors_sym(A), eigenvalues_sym(A))` but with a lower computational cost due to the shared work between the two results.

Available since 2.33

Because multiplying an eigenvector by -1 results in an eigenvector, eigenvectors returned by a decomposition are only identified up to a sign change. In order to compare the eigenvectors produced by Stan's eigendecomposition to others, signs may need to be normalized in some way, such as by fixing the sign of a component, or doing comparisons allowing a multiplication by -1.

The condition number of a symmetric matrix is defined to be the ratio of the largest eigenvalue to the smallest eigenvalue. Large condition numbers lead to difficulty in numerical algorithms such as computing inverses, and thus known as "ill conditioned." The ratio can even be infinite in the case of singular matrices (i.e., those with eigenvalues of 0).

QR decomposition

`matrix` **`qr_thin_Q`**`(matrix A)`
The orthogonal matrix in the thin QR decomposition of A, which implies that the resulting matrix has the same dimensions as A

Available since 2.18

`matrix` **`qr_thin_R`**`(matrix A)`
The upper triangular matrix in the thin QR decomposition of A, which implies that the resulting matrix is square with the same number of columns as A

Available since 2.18

`tuple(matrix, matrix)` **`qr_thin`**`(matrix A)`
Returns both portions of the QR decomposition of A. The first element ("Q") is the orthonormal matrix in the thin QR decomposition and the second element ("R") is upper triangular. This function is equivalent to `(qr_thin_Q(A), qr_thin_R(A))` but with a lower computational cost due to the shared work between the two results.

Available since 2.33

`matrix` **`qr_Q`**`(matrix A)`
The orthogonal matrix in the fat QR decomposition of A, which implies that the resulting matrix is square with the same number of rows as A

Available since 2.3

`matrix` **`qr_R`**`(matrix A)`
The upper trapezoidal matrix in the fat QR decomposition of A, which implies that the resulting matrix will be rectangular with the same dimensions as A

Available since 2.3

`tuple(matrix, matrix)` **`qr`**`(matrix A)`
Returns both portions of the QR decomposition of A. The first element ("Q") is the orthonormal matrix in the thin QR decomposition and the second element ("R") is upper triangular. This function is equivalent to `(qr_Q(A), qr_R(A))` but with a lower computational cost due to the shared work between the two results.

Available since 2.33

The thin QR decomposition is always preferable because it will consume much less memory when the input matrix is large than will the fat QR decomposition. Both versions of the decomposition represent the input matrix as

$$A = Q\,R.$$

Multiplying a column of an orthogonal matrix by -1 still results in an orthogonal matrix, and you can multiply the corresponding row of the upper trapezoidal matrix by -1 without changing the product. Thus, Stan adopts the normalization that the diagonal elements of the upper trapezoidal matrix are strictly positive and the columns of the orthogonal matrix are reflected if necessary. Also, these QR decomposition algorithms do not utilize pivoting and thus may be numerically unstable on input matrices that have less than full rank.

Cholesky decomposition

Every symmetric, positive-definite matrix (such as a correlation or covariance matrix) has a Cholesky decomposition. If Σ is a symmetric, positive-definite matrix, its Cholesky decomposition is the lower-triangular vector L such that

$$\Sigma = L\,L^{\top}.$$

`matrix` **`cholesky_decompose`**`(matrix A)`
The lower-triangular Cholesky factor of the symmetric positive-definite matrix A

Available since 2.0

Singular value decomposition

The matrix A can be decomposed into a diagonal matrix of singular values, D, and matrices of its left and right singular vectors, U and V,

$$A = UDV^T.$$

The matrices of singular vectors here are thin. That is for an N by P input A, $M = min(N, P)$, U is size N by M and V is size P by M.

`vector` **`singular_values`**`(matrix A)`
The singular values of `A` in descending order

Available since 2.0

`matrix` **`svd_U`**`(matrix A)`
The left-singular vectors of `A`

Available since 2.26

`matrix` **`svd_V`**`(matrix A)`
The right-singular vectors of `A`

Available since 2.26

`tuple(matrix, vector, matrix)` **`svd`**`(matrix A)`
Returns a tuple containing the left-singular vectors of `A`, the singular values of `A` in descending order, and the right-singular values of `A`. This function is equivalent to `(svd_U(A), singular_values(A), svd_V(A))` but with a lower computational cost due to the shared work between the different components.

Available since 2.33

6.15. Sort functions

See the sorting functions section for examples of how the functions work.

`vector` **`sort_asc`**`(vector v)`
Sort the elements of v in ascending order

Available since 2.0

`row_vector` **`sort_asc`**`(row_vector v)`
Sort the elements of v in ascending order

Available since 2.0

`vector` **`sort_desc`**`(vector v)`
Sort the elements of v in descending order

Available since 2.0

`row_vector` **`sort_desc`**`(row_vector v)`
Sort the elements of v in descending order

Available since 2.0

`array[] int` **`sort_indices_asc`**`(vector v)`
Return an array of indices between 1 and the size of v, sorted to index v in ascending order.

Available since 2.3

`array[] int` **`sort_indices_asc`**`(row_vector v)`
Return an array of indices between 1 and the size of v, sorted to index v in ascending order.

Available since 2.3

`array[] int` **`sort_indices_desc`**`(vector v)`
Return an array of indices between 1 and the size of v, sorted to index v in descending order.

Available since 2.3

`array[] int` **`sort_indices_desc`**`(row_vector v)`
Return an array of indices between 1 and the size of v, sorted to index v in descending order.

Available since 2.3

`int` **`rank`**`(vector v, int s)`
Number of components of v less than v[s]

Available since 2.0

`int` **`rank`**`(row_vector v, int s)`
Number of components of v less than v[s]

Available since 2.0

6.16. Reverse functions

`vector` **`reverse`**`(vector v)`
Return a new vector containing the elements of the argument in reverse order.

Available since 2.23

`row_vector` **`reverse`**`(row_vector v)`
Return a new row vector containing the elements of the argument in reverse order.

Available since 2.23

7. Complex Matrix Operations

7.1. Complex promotion

This chapter provides the details of functions that operate over complex matrices, vectors, and row vectors. These mirror the operations over real complex_matrix types and are defined in the usual way for complex numbers.

Promotion of complex arguments

If an expression `e` can be assigned to a variable of type `T`, then it can be used as an argument to a function that is specified to take arguments of type `T`. For instance, `sqrt(real)` is specified to take a `real` argument, but an integer expression such as `2 + 2` of type `int` can be passed to `sqrt`, so that `sqrt(2 + 2)` is well defined. This works by promoting the integer expression `2 + 2` to be of `real` type.

The rules for promotion in Stan are simple:

- `int` may be promoted to `real`,
- `real` may be promoted to `complex`,
- `vector` can be promoted to `complex_vector`,
- `row_vector` can be promoted to `complex_row_vector`,
- `matrix` can be promoted to `complex_matrix`,
- if `T` can be promoted to `U` and `U` can be promoted to `V`, then `T` can be promoted to `V` (transitive), and
- if `T` can be promoted to `U`, then `T[]` can be promoted to `U[]` (covariant).

Signature selection

When a function is called, the definition requiring the fewest number of promotions is used. For example, when calling `vector + vector`, the real-valued signature is used. When calling any of `complex_vector + vector`, `vector + complex_vector`, or `complex_vector + complex_vector`, the complex signature is used. If more than one signature matches with a the minimal number of promotions, the call is ambiguous, and an error will be raised by the compiler. Promotion ambiguity leading to ill-defined calls should never happen with Stan built-in functions.

Signatures for complex functions

Complex function signatures will only list the fully complex type. For example, with complex vector addition, we will list a single signature, `complex operator+(complex_vector, complex_vector)`. Through promotion, `operator+` may be called with one complex vector and one real vector as well, but the documentation elides the implied signatures `operator+(complex_vector, vector)` and `operator+(vector, complex_vector)`.

Generic functions work for complex containers

Generic functions work for arrays containing complex, complex matrix, complex vector, or complex row vector types. This includes the functions `append_array`, `dims`, `head`, `num_elements`, `rep_array`, `reverse`, `segment`, `size`, and `tail`.

7.2. Integer-valued complex matrix size functions

`int` **`num_elements`**`(complex_vector x)`
The total number of elements in the vector x (same as function `rows`)

Available since 2.30

`int` **`num_elements`**`(complex_row_vector x)`
The total number of elements in the vector x (same as function `cols`)

Available since 2.30

`int` **`num_elements`**`(complex_matrix x)`
The total number of elements in the matrix x. For example, if `x` is a 5×3 matrix, then `num_elements(x)` is 15

Available since 2.30

`int` **`rows`**`(complex_vector x)`
The number of rows in the vector x

Available since 2.30

`int` **`rows`**`(complex_row_vector x)`
The number of rows in the row vector x, namely 1

Available since 2.30

`int` **`rows`**`(complex_matrix x)`
The number of rows in the matrix x

Available since 2.30

`int` **`cols`**`(complex_vector x)`
The number of columns in the vector x, namely 1

Available since 2.30

`int` **`cols`**`(complex_row_vector x)`
The number of columns in the row vector x

Available since 2.30

`int` **`cols`**`(complex_matrix x)`
The number of columns in the matrix x

Available since 2.30

`int` **`size`**`(complex_vector x)`
The size of `x`, i.e., the number of elements

Available since 2.30

`int` **`size`**`(complex_row_vector x)`
The size of `x`, i.e., the number of elements

Available since 2.30

`int` **`size`**`(matrix x)`
The size of the matrix `x`. For example, if `x` is a 5×3 matrix, then `size(x)` is 15.

Available since 2.30

7.3. Complex matrix arithmetic operators

Stan supports all basic complex arithmetic operators using infix, prefix and postfix operations. This section lists the operations supported by Stan along with their argument and result types.

Negation prefix operators

`complex_vector` **`operator-`**`(complex_vector x)`
The negation of the vector x.

Available since 2.30

`complex_row_vector` **`operator-`**`(complex_row_vector x)`
The negation of the row vector x.

Available since 2.30

`complex_matrix` **`operator-`**`(complex_matrix x)`
The negation of the matrix x.

Available since 2.30

`T` **`operator-`**`(T x)`
Vectorized version of `operator-`. If `T x` is a (possibly nested) array of matrix types, `-x` is the same shape array where each individual value is negated.

Available since 2.31

Infix complex_matrix operators

`complex_vector` **`operator+`**`(complex_vector x, complex_vector y)`
The sum of the vectors x and y.

Available since 2.30

`complex_row_vector` **`operator+`**`(complex_row_vector x, complex_row_vector y)`
The sum of the row vectors x and y.

Available since 2.30

`complex_matrix` **`operator+`**`(complex_matrix x, complex_matrix y)`
The sum of the matrices x and y

Available since 2.30

`complex_vector` **`operator-`**`(complex_vector x, complex_vector y)`
The difference between the vectors x and y.

Available since 2.30

`complex_row_vector` **`operator-`**`(complex_row_vector x, complex_row_vector y)`
The difference between the row vectors x and y

Available since 2.30

`complex_matrix` **`operator-`**`(complex_matrix x, complex_matrix y)`
The difference between the matrices x and y

Available since 2.30

`complex_vector` **`operator*`**`(complex x, complex_vector y)`
The product of the scalar x and vector y

Available since 2.30

`complex_row_vector` **`operator*`**`(complex x, complex_row_vector y)`
The product of the scalar x and the row vector y

Available since 2.30

`complex_matrix` **`operator*`**`(complex x, complex_matrix y)`
The product of the scalar x and the matrix y

Available since 2.30

`complex_vector` **`operator*`**`(complex_vector x, complex y)`
The product of the scalar y and vector x

Available since 2.30

`complex_matrix` **`operator*`**`(complex_vector x, complex_row_vector y)`
The product of the vector x and row vector y

Available since 2.30

`complex_row_vector` **`operator*`**`(complex_row_vector x, complex y)`
The product of the scalar y and row vector x

Available since 2.30

`complex` **`operator*`**`(complex_row_vector x, complex_vector y)`
The product of the row vector x and vector y

Available since 2.30

`complex_row_vector` **`operator*`**`(complex_row_vector x, complex_matrix y)`
The product of the row vector x and matrix y

Available since 2.30

`complex_matrix` **`operator*`**`(complex_matrix x, complex y)`
The product of the scalar y and matrix x

Available since 2.30

`complex_vector` **`operator*`**`(complex_matrix x, complex_vector y)`
The product of the matrix x and vector y

Available since 2.30

`complex_matrix` **`operator*`**`(complex_matrix x, complex_matrix y)`
The product of the matrices x and y

Available since 2.30

Broadcast infix operators

`complex_vector` **`operator+`**`(complex_vector x, complex y)`
The result of adding y to every entry in the vector x

Available since 2.30

`complex_vector` **`operator+`**`(complex x, complex_vector y)`
The result of adding x to every entry in the vector y

Available since 2.30

`complex_row_vector` **`operator+`**`(complex_row_vector x, complex y)`
The result of adding y to every entry in the row vector x

Available since 2.30

`complex_row_vector` **`operator+`**`(complex x, complex_row_vector y)`
The result of adding x to every entry in the row vector y

Available since 2.30

`complex_matrix` **`operator+`**`(complex_matrix x, complex y)`
The result of adding y to every entry in the matrix x

Available since 2.30

`complex_matrix` **`operator+`**`(complex x, complex_matrix y)`
The result of adding x to every entry in the matrix y

Available since 2.30

`complex_vector` **`operator-`**`(complex_vector x, complex y)`
The result of subtracting y from every entry in the vector x

Available since 2.30

`complex_vector` **`operator-`**`(complex x, complex_vector y)`
The result of adding x to every entry in the negation of the vector y

Available since 2.30

`complex_row_vector` **`operator-`**`(complex_row_vector x, complex y)`
The result of subtracting y from every entry in the row vector x

Available since 2.30

`complex_row_vector` **`operator-`**`(complex x, complex_row_vector y)`
The result of adding x to every entry in the negation of the row vector y

Available since 2.30

`complex_matrix` **`operator-`**`(complex_matrix x, complex y)`
The result of subtracting y from every entry in the matrix x

Available since 2.30

`complex_matrix` **`operator-`**`(complex x, complex_matrix y)`
The result of adding x to every entry in negation of the matrix y

Available since 2.30

`complex_vector` **`operator/`**`(complex_vector x, complex y)`
The result of dividing each entry in the vector x by y

Available since 2.30

`complex_row_vector` **`operator/`**`(complex_row_vector x, complex y)`
The result of dividing each entry in the row vector x by y

Available since 2.30

`complex_matrix` **`operator/`**`(complex_matrix x, complex y)`
The result of dividing each entry in the matrix x by y

Available since 2.30

7.4. Complex Transposition Operator

Complex complex_matrix transposition is represented using a postfix operator.

`complex_matrix` **`operator'`**`(complex_matrix x)`
The transpose of the matrix `x`, written as `x'`

Available since 2.30

`complex_row_vector` **`operator'`**`(complex_vector x)`
The transpose of the vector `x`, written as `x'`

Available since 2.30

`complex_vector` **`operator'`**`(complex_row_vector x)`
The transpose of the row vector `x`, written as `x'`

Available since 2.30

7.5. Complex elementwise functions

As in the real case, elementwise complex functions apply a function to each element of a vector or matrix, returning a result of the same shape as the argument.

`complex_vector` **`operator.*`**`(complex_vector x, complex_vector y)`
The elementwise product of `x` and `y`

Available since 2.30

`complex_row_vector` **`operator.*`**`(complex_row_vector x, complex_row_vector y)`
The elementwise product of `x` and `y`

Available since 2.30

`complex_matrix` **`operator.*`**`(complex_matrix x, complex_matrix y)`
The elementwise product of `x` and `y`

Available since 2.30

`complex_vector` **`operator./`**`(complex_vector x, complex_vector y)`
The elementwise quotient of `x` and `y`

Available since 2.30

`complex_vector` **`operator./`**`(complex x, complex_vector y)`
The elementwise quotient of `x` and `y`

Available since 2.30

`complex_vector` **`operator./`**`(complex_vector x, complex y)`
The elementwise quotient of `x` and `y`

Available since 2.30

`complex_row_vector` **`operator./`**`(complex_row_vector x, complex_row_vector y)`
The elementwise quotient of `x` and `y`

Available since 2.30

`complex_row_vector` **`operator./`**`(complex x, complex_row_vector y)`
The elementwise quotient of `x` and `y`

Available since 2.30

`complex_row_vector` **`operator./`**`(complex_row_vector x, complex y)`
The elementwise quotient of `x` and `y`

Available since 2.30

`complex_matrix` **`operator./`**`(complex_matrix x, complex_matrix y)`
The elementwise quotient of `x` and `y`

Available since 2.30

`complex_matrix` **`operator./`**`(complex x, complex_matrix y)`
The elementwise quotient of `x` and `y`

Available since 2.30

`complex_matrix` **`operator./`**`(complex_matrix x, complex y)`
The elementwise quotient of `x` and `y`

Available since 2.30

`vector` **`operator.^`**`(complex_vector x, complex_vector y)`
The elementwise power of y and x

Available since 2.30

`vector` **`operator.^`**`(complex_vector x, complex y)`
The elementwise power of y and x

Available since 2.30

`vector` **`operator.^`**`(complex x, complex_vector y)`
The elementwise power of y and x

Available since 2.30

`row_vector` **`operator.^`**`(complex_row_vector x, complex_row_vector y)`
The elementwise power of y and x

Available since 2.30

`row_vector` **`operator.^`**`(complex_row_vector x, complex y)`
The elementwise power of y and x

Available since 2.30

`row_vector` **`operator.^`**`(complex x, complex_row_vector y)`
The elementwise power of y and x

Available since 2.30

`matrix` **`operator.^`**`( complex_matrix x, complex_matrix y)`
The elementwise power of y and x

Available since 2.30

`matrix` **`operator.^`**`( complex_matrix x, complex y)`
The elementwise power of y and x

Available since 2.30

`matrix` **`operator.^`**`(complex x, complex_matrix y)`
The elementwise power of y and x

Available since 2.30

7.6. Dot products and specialized products for complex matrices

`complex` **`dot_product`**`(complex_vector x, complex_vector y)`
The dot product of x and y

Available since 2.30

`complex` **`dot_product`**`(complex_vector x, complex_row_vector y)`
The dot product of x and y

Available since 2.30

`complex` **`dot_product`**`(complex_row_vector x, complex_vector y)`
The dot product of x and y

Available since 2.30

`complex` **`dot_product`**`(complex_row_vector x, complex_row_vector y)`
The dot product of x and y

Available since 2.30

`complex_row_vector` **`columns_dot_product`**`(complex_vector x, complex_vector y)`
The dot product of the columns of x and y

Available since 2.30

`complex_row_vector` **`columns_dot_product`**`(complex_row_vector x, complex_row_vector y)`
The dot product of the columns of x and y

Available since 2.30

`complex_row_vector` **`columns_dot_product`**`(complex_matrix x, complex_matrix y)`

The dot product of the columns of x and y

Available since 2.30

`complex_vector` **`rows_dot_product`**`(complex_vector x, complex_vector y)`
The dot product of the rows of x and y

Available since 2.30

`complex_vector` **`rows_dot_product`**`(complex_row_vector x, complex_row_vector y)`
The dot product of the rows of x and y

Available since 2.30

`complex_vector` **`rows_dot_product`**`(complex_matrix x, complex_matrix y)`
The dot product of the rows of x and y

Available since 2.30

`complex` **`dot_self`**`(complex_vector x)`
The dot product of the vector x with itself

Available since 2.30

`complex` **`dot_self`**`(complex_row_vector x)`
The dot product of the row vector x with itself

Available since 2.30

`complex_row_vector` **`columns_dot_self`**`(complex_vector x)`
The dot product of the columns of x with themselves

Available since 2.30

`complex_row_vector` **`columns_dot_self`**`(complex_row_vector x)`
The dot product of the columns of x with themselves

Available since 2.30

`complex_row_vector` **`columns_dot_self`**`(complex_matrix x)`
The dot product of the columns of x with themselves

Available since 2.30

`complex_vector` **`rows_dot_self`**`(complex_vector x)`
The dot product of the rows of x with themselves

Available since 2.30

`complex_vector` **`rows_dot_self`**`(complex_row_vector x)`
The dot product of the rows of x with themselves

Available since 2.30

`complex_vector` **`rows_dot_self`**`(complex_matrix x)`
The dot product of the rows of x with themselves

Available since 2.30

Specialized products

`complex_matrix` **`diag_pre_multiply`**`(complex_vector v, complex_matrix m)`
Return the product of the diagonal matrix formed from the vector v and the matrix m, i.e., `diag_matrix(v) * m`.

Available since 2.30

`complex_matrix` **`diag_pre_multiply`**`(complex_row_vector v, complex_matrix m)`
Return the product of the diagonal matrix formed from the vector rv and the matrix m, i.e., `diag_matrix(rv) * m`.

Available since 2.30

`complex_matrix` **`diag_post_multiply`**`(complex_matrix m, complex_vector v)`
Return the product of the matrix m and the diagonal matrix formed from the vector v, i.e., `m * diag_matrix(v)`.

Available since 2.30

`complex_matrix` **`diag_post_multiply`**`(complex_matrix m, complex_row_vector v)`
Return the product of the matrix `m` and the diagonal matrix formed from the the row vector `rv`, i.e., `m * diag_matrix(rv)`.

Available since 2.30

7.7. Complex reductions

Sums and products

`complex` **`sum`**`(complex_vector x)`
The sum of the values in x, or 0 if x is empty

Available since 2.30

`complex` **`sum`**`(complex_row_vector x)`
The sum of the values in x, or 0 if x is empty

Available since 2.30

`complex` **`sum`**`(complex_matrix x)`
The sum of the values in x, or 0 if x is empty

Available since 2.30

`complex` **`prod`**`(complex_vector x)`
The product of the values in x, or 1 if x is empty

Available since 2.30

`complex` **`prod`**`(complex_row_vector x)`
The product of the values in x, or 1 if x is empty

Available since 2.30

`complex` **`prod`**`(complex_matrix x)`
The product of the values in x, or 1 if x is empty

Available since 2.30

7.8. Vectorized accessor functions

Much like with complex scalars, two functions are defined to get the real and imaginary components of complex-valued objects.

Type "demotion"

These functions return the same shape (e.g., matrix, vector, row vector, or array) object as their input, but demoted to a `real` type. For example, `get_real(complex_matrix M)` yields a `matrix` containing the real component of each value in `M`.

The following table contains examples of what this notation can mean:

Type `T`	Type `T_demoted`
`complex`	`real`
`complex_vector`	`vector`
`complex_row_vector`	`row_vector`
`complex_matrix`	`matrix`
`array[] complex`	`array[] real`
`array[,,] complex`	`array[,,] real`

Real and imaginary component accessor functions

`T_demoted` **`get_real`**`(T x)`
Given an object of complex type `T`, return the same shape object but of type `real` by getting the real component of each element of `x`.

Available since 2.30

`T_demoted` **`get_imag`**`(T x)`
Given an object of complex type `T`, return the same shape object but of type `real` by getting the imaginary component of each element of `x`.

Available since 2.30

For example, given the Stan declaration

```
complex_vector[2] z = [3+4i, 5+6i]';
```

A call `get_real(z)` will yield the vector `[3, 5]'`, and a call `get_imag(z)` will yield the vector `[4, 6]'`.

7.9. Complex broadcast functions

The following broadcast functions allow vectors, row vectors and matrices to be created by copying a single element into all of their cells. Matrices may also be created by stacking copies of row vectors vertically or stacking copies of column vectors horizontally.

`complex_vector` **`rep_vector`**`(complex z, int m)`
Return the size m (column) vector consisting of copies of z.

Available since 2.30

`complex_row_vector` **`rep_row_vector`**`(complex z, int n)`
Return the size n row vector consisting of copies of z.

Available since 2.30

`complex_matrix` **`rep_matrix`**`(complex z, int m, int n)`
Return the m by n matrix consisting of copies of z.

Available since 2.30

`complex_matrix` **`rep_matrix`**`(complex_vector v, int n)`
Return the m by n matrix consisting of n copies of the (column) vector v of size m.

Available since 2.30

`complex_matrix` **`rep_matrix`**`(complex_row_vector rv, int m)`
Return the m by n matrix consisting of m copies of the row vector rv of size n.

Available since 2.30

Symmetrization

`complex_matrix` **`symmetrize_from_lower_tri`**`(complex_matrix A)`
Construct a symmetric matrix from the lower triangle of A.

Available since 2.30

7.10. Diagonal complex matrix functions

`complex_matrix` **`add_diag`**`(complex_matrix m, complex_row_vector d)`
Add row_vector `d` to the diagonal of matrix `m`.

Available since 2.30

`complex_matrix` **`add_diag`**`(complex_matrix m, complex_vector d)`
Add vector `d` to the diagonal of matrix `m`.

Available since 2.30

`complex_matrix` **`add_diag`**`(complex_matrix m, complex d)`
Add scalar `d` to every diagonal element of matrix `m`.

Available since 2.30

`complex_vector` **`diagonal`**`(complex_matrix x)`
The diagonal of the matrix x

Available since 2.30

`complex_matrix` **`diag_matrix`**`(complex_vector x)`
The diagonal matrix with diagonal x

Available since 2.30

7.11. Slicing and blocking functions for complex matrices

Stan provides several functions for generating slices or blocks or diagonal entries for matrices.

Columns and rows

`complex_vector` **`col`**`(complex_matrix x, int n)`
The n-th column of matrix x

Available since 2.30

`complex_row_vector` **`row`**`(complex_matrix x, int m)`
The m-th row of matrix x

Available since 2.30

Block operations

Matrix slicing operations

`complex_matrix` **`block`**`(complex_matrix x, int i, int j, int n_rows, int n_cols)`
Return the submatrix of x that starts at row i and column j and extends n_rows rows and n_cols columns.

Available since 2.30

`complex_vector` **`sub_col`**`(complex_matrix x, int i, int j, int n_rows)`
Return the sub-column of x that starts at row i and column j and extends n_rows rows and 1 column.

Available since 2.30

`complex_row_vector` **`sub_row`**`(complex_matrix x, int i, int j, int n_cols)`
Return the sub-row of x that starts at row i and column j and extends 1 row and n_cols columns.

Available since 2.30

Vector slicing operations.

`complex_vector` **`head`**`(complex_vector v, int n)`
Return the vector consisting of the first n elements of v.

Available since 2.30

`complex_row_vector` **`head`**`(complex_row_vector rv, int n)`
Return the row vector consisting of the first n elements of rv.

Available since 2.30

`complex_vector` **`tail`**`(complex_vector v, int n)`
Return the vector consisting of the last n elements of v.

Available since 2.30

`complex_row_vector` **`tail`**`(complex_row_vector rv, int n)`
Return the row vector consisting of the last n elements of rv.

Available since 2.30

`complex_vector` **`segment`**`(complex_vector v, int i, int n)`
Return the vector consisting of the n elements of v starting at i; i.e., elements i through through i + n - 1.

Available since 2.30

`complex_row_vector` **`segment`**`(complex_row_vector rv, int i, int n)`
Return the row vector consisting of the n elements of rv starting at i; i.e., elements i through through i + n - 1.

Available since 2.30

7.12. Complex matrix concatenation

Horizontal concatenation

`complex_matrix` **`append_col`**`(complex_matrix x, complex_matrix y)`
Combine matrices x and y by column. The matrices must have the same number of rows.

Available since 2.30

`complex_matrix` **`append_col`**`(complex_matrix x, complex_vector y)`
Combine matrix x and vector y by column. The matrix and the vector must have the same number of rows.

Available since 2.30

`complex_matrix` **`append_col`**`(complex_vector x, complex_matrix y)`
Combine vector x and matrix y by column. The vector and the matrix must have the same number of rows.

Available since 2.30

`complex_matrix` **`append_col`**`(complex_vector x, complex_vector y)`
Combine vectors x and y by column. The vectors must have the same number of rows.

Available since 2.30

`complex_row_vector` **`append_col`**`(complex_row_vector x, complex_row_vector y)`
Combine row vectors x and y (of any size) into another row vector by appending y to the end of x.

Available since 2.30

`complex_row_vector` **`append_col`**`(complex x, complex_row_vector y)`
Append x to the front of y, returning another row vector.

Available since 2.30

`complex_row_vector` **`append_col`**`(complex_row_vector x, complex y)`
Append y to the end of x, returning another row vector.

Available since 2.30

Vertical concatenation

`complex_matrix` **`append_row`**`(complex_matrix x, complex_matrix y)`
Combine matrices x and y by row. The matrices must have the same number of columns.

Available since 2.30

`complex_matrix` **`append_row`**`(complex_matrix x, complex_row_vector y)`
Combine matrix x and row vector y by row. The matrix and the row vector must have the same number of columns.

Available since 2.30

`complex_matrix` **`append_row`**`(complex_row_vector x, complex_matrix y)`
Combine row vector x and matrix y by row. The row vector and the matrix must have the same number of columns.

Available since 2.30

`complex_matrix` **`append_row`**`(complex_row_vector x, complex_row_vector y)`
Combine row vectors x and y by row. The row vectors must have the same number of columns.

Available since 2.30

`complex_vector` **`append_row`**`(complex_vector x, complex_vector y)`
Concatenate vectors x and y of any size into another vector.

Available since 2.30

`complex_vector` **`append_row`**`(complex x, complex_vector y)`
Append x to the top of y, returning another vector.

Available since 2.30

`complex_vector` **`append_row`**`(complex_vector x, complex y)`
Append y to the bottom of x, returning another vector.

Available since 2.30

7.13. Complex special matrix functions

Fast Fourier transforms

Stan's fast Fourier transform functions take the standard definition of the discrete Fourier transform (see the definitions below for specifics) and scale the inverse transform by one over dimensionality so that the following identities hold for complex vectors `u` and `v`,

```
fft(inv_fft(u)) == u        inv_fft(fft(v)) == v
```

and in the 2-dimensional case for complex matrices `A` and `B`,

```
fft2(inv_fft2(A)) == A      inv_fft2(fft2(B)) == B
```

Although the FFT functions only accept complex inputs, real vectors and matrices will be promoted to their complex counterparts before applying the FFT functions.

`complex_vector` **`fft`**`(complex_vector v)`
Return the discrete Fourier transform of the specified complex vector `v`. If $v \in \mathbb{C}^N$ is a complex vector with N elements and $u = \text{fft}(v)$, then

$$u_n = \sum_{m<n} v_m \cdot \exp\left(\frac{-n \cdot m \cdot 2 \cdot \pi \cdot \sqrt{-1}}{N}\right).$$

Available since 2.30

`complex_matrix` **`fft2`**`(complex_matrix m)`
Return the 2D discrete Fourier transform of the specified complex matrix `m`. The 2D FFT is defined as the result of applying the FFT to each row and then to each column.

Available since 2.30

`complex_vector` **`inv_fft`**`(complex_vector u)`
Return the inverse of the discrete Fourier transform of the specified complex vector `u`. The inverse FFT (this function) is scaled so that `fft(inv_fft(u)) == u`. If $u \in \mathbb{C}^N$ is a complex vector with N elements and $v = \text{fft}^{-1}(u)$, then

$$v_n = \frac{1}{N} \sum_{m<n} u_m \cdot \exp\left(\frac{n \cdot m \cdot 2 \cdot \pi \cdot \sqrt{-1}}{N}\right).$$

This only differs from the FFT by the sign inside the exponential and the scaling. The $\frac{1}{N}$ scaling ensures that `fft(inv_fft(u)) == u` and `inv_fft(fft(v)) == v` for complex vectors `u` and `v`.

Available since 2.30

`complex_matrix` **`inv_fft2`**`(complex_matrix m)`
Return the inverse of the 2D discrete Fourier transform of the specified complex matrix `m`. The 2D inverse FFT is defined as the result of applying the inverse FFT to each row and then to each column. The invertible scaling of the inverse FFT ensures `fft2(inv_fft2(A)) == A` and `inv_fft2(fft2(B)) == B`.

Available since 2.30

Cumulative sums

The cumulative sum of a sequence $x_1, \ldots, x_N$ is the sequence $y_1, \ldots, y_N$, where

$$y_n = \sum_{m=1}^{n} x_m.$$

`array[] complex` **`cumulative_sum`**`(array[] complex x)`
The cumulative sum of x

Available since 2.30

`complex_vector` **`cumulative_sum`**`(complex_vector v)`
The cumulative sum of v

Available since 2.30

`complex_row_vector` **`cumulative_sum`**`(complex_row_vector rv)`
The cumulative sum of rv

Available since 2.30

7.14. Complex linear algebra functions

Complex matrix division operators and functions

In general, it is much more efficient and also more arithmetically stable to use matrix division than to multiply by an inverse.

Complex matrix division operators

`complex_row_vector` **`operator/`**`(complex_row_vector b, complex_matrix A)`
The right division of b by A; equivalently `b * inverse(A)`

Available since 2.30

`complex_matrix` **`operator/`**`(complex_matrix B, complex_matrix A)`
The right division of B by A; equivalently `B * inverse(A)`

Available since 2.30

Linear algebra functions

Trace

`complex` **`trace`**`(complex_matrix A)`
The trace of A, or 0 if A is empty; A is not required to be diagonal

Available since 2.30

Eigendecomposition

`complex_vector` **`eigenvalues`**`(complex_matrix A)`
The complex-valued vector of eigenvalues of the matrix `A`. The eigenvalues are repeated according to their algebraic multiplicity, so there are as many eigenvalues as rows in the matrix. The eigenvalues are not sorted in any particular order.

Available since 2.32

`complex_matrix` **`eigenvectors`**`(complex_matrix A)`
The matrix with the complex-valued (column) eigenvectors of the matrix `A` in the same order as returned by the function `eigenvalues`

Available since 2.32

`tuple(complex_matrix, complex_vector)` **`eigendecompose`**`(complex_matrix A)`
Return the matrix of (column) eigenvectors and vector of eigenvalues of the matrix `A`. This function is equivalent to `(eigenvectors(A), eigenvalues(A))` but with a lower computational cost due to the shared work between the two results.

Available since 2.33

`complex_vector` **`eigenvalues_sym`**`(complex_matrix A)`
The vector of eigenvalues of a symmetric matrix `A` in ascending order

Available since 2.30

`complex_matrix` **`eigenvectors_sym`**`(complex_matrix A)`
The matrix with the (column) eigenvectors of symmetric matrix `A` in the same order as returned by the function `eigenvalues_sym`

Available since 2.30

`tuple(complex_matrix, complex_vector)` **`eigendecompose_sym`**`(complex_matrix A)`
Return the matrix of (column) eigenvectors and vector of eigenvalues of the symmetric matrix `A`. This function is equivalent to `(eigenvectors_sym(A), eigenvalues_sym(A))` but with a lower computational cost due to the shared work between the two results.

Available since 2.33

Because multiplying an eigenvector by -1 results in an eigenvector, eigenvectors returned by a decomposition are only identified up to a sign change. In order to compare the eigenvectors produced by Stan's eigendecomposition to others, signs may need to be normalized in some way, such as by fixing the sign of a component, or doing comparisons allowing a multiplication by -1.

The condition number of a symmetric matrix is defined to be the ratio of the largest eigenvalue to the smallest eigenvalue. Large condition numbers lead to difficulty in numerical algorithms such as computing inverses, and thus known as "ill conditioned." The ratio can even be infinite in the case of singular matrices (i.e., those with eigenvalues of 0).

Singular value decomposition

The matrix A can be decomposed into a diagonal matrix of singular values, D, and matrices of its left and right singular vectors, U and V,

$$A = UDV^T.$$

The matrices of singular vectors here are thin. That is for an N by P input A, $M = min(N, P)$, U is size N by M and V is size P by M.

`vector` **`singular_values`**`(complex_matrix A)`
The singular values of `A` in descending order

Available since 2.30

`complex_matrix` **`svd_U`**`(complex_matrix A)`
The left-singular vectors of `A`

Available since 2.30

`complex_matrix` **`svd_V`**`(complex_matrix A)`
The right-singular vectors of `A`

Available since 2.30

`tuple(complex_matrix, vector, complex_matrix)` **`svd`**`(complex_matrix A)`
Returns a tuple containing the left-singular vectors of `A`, the singular values of `A` in descending order, and the right-singular values of `A`. This function is equivalent to `(svd_U(A), singular_values(A), svd_V(A))` but with a lower computational cost due to the shared work between the different components.

Available since 2.33

Complex Schur Decomposition

The complex Schur decomposition of a square matrix A produces a complex unitary matrix U and a complex upper-triangular Schur form matrix T such that

$$A = U \cdot T \cdot U^{-1}$$

Since U is unitary, its inverse is also its conjugate transpose, $U^{-1} = U^*$, $U^*(i,j) = \mathrm{conj}(U(j,i))$

`complex_matrix` **`complex_schur_decompose_t`**`(matrix A)`
Compute the upper-triangular Schur form matrix of the complex Schur decomposition of `A`.

Available since 2.31

`complex_matrix` **`complex_schur_decompose_t`**`(complex_matrix A)`
Compute the upper-triangular Schur form matrix of the complex Schur decomposition of `A`.

Available since 2.31

`complex_matrix` **`complex_schur_decompose_u`**`(matrix A)`
Compute the unitary matrix of the complex Schur decomposition of `A`.

Available since 2.31

`complex_matrix` **`complex_schur_decompose_u`**`(complex_matrix A)`
Compute the unitary matrix of the complex Schur decomposition of `A`.

Available since 2.31

`tuple(complex_matrix, complex_matrix)` **`complex_schur_decompose`**`(matrix A)`
Returns the unitary matrix and the upper-triangular Schur form matrix of the complex Schur decomposition of `A`. This function is equivalent to `(complex_schur_decompose_u(A), complex_schur_decompose_t(A))` but with a lower computational cost due to the shared work between the two results. This

overload is equivalent to `complex_schur_decompose(to_complex(A,0))` but is more efficient.

Available since 2.33

`tuple(complex_matrix, complex_matrix)` **`complex_schur_decompose`**`(complex_matrix A)`
Returns the unitary matrix and the upper-triangular Schur form matrix of the complex Schur decomposition of `A`. This function is equivalent to `(complex_schur_decompose_u(A), complex_schur_decompose_t(A))` but with a lower computational cost due to the shared work between the two results.

Available since 2.33

7.15. Reverse functions for complex matrices

`complex_vector` **`reverse`**`(complex_vector v)`
Return a new vector containing the elements of the argument in reverse order.

Available since 2.30

`complex_row_vector` **`reverse`**`(complex_row_vector v)`
Return a new row vector containing the elements of the argument in reverse order.

Available since 2.30

8. Sparse Matrix Operations

For sparse matrices, for which many elements are zero, it is more efficient to use specialized representations to save memory and speed up matrix arithmetic (including derivative calculations). Given Stan's implementation, there is substantial space (memory) savings by using sparse matrices. Because of the ease of optimizing dense matrix operations, speed improvements only arise at 90% or even greater sparsity; below that level, dense matrices are faster but use more memory.

Because of this speedup and space savings, it may even be useful to read in a dense matrix and convert it to a sparse matrix before multiplying it by a vector. This chapter covers a very specific form of sparsity consisting of a sparse matrix multiplied by a dense vector.

8.1. Compressed row storage

Sparse matrices are represented in Stan using compressed row storage (CSR). For example, the matrix

$$A = \begin{bmatrix} 19 & 27 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 52 \\ 81 & 0 & 95 & 33 \end{bmatrix}$$

is translated into a vector of the non-zero real values, read by row from the matrix A,

$$w(A) = \begin{bmatrix} 19 & 27 & 52 & 81 & 95 & 33 \end{bmatrix}^\top,$$

an array of integer column indices for the values,

$$v(A) = \begin{bmatrix} 1 & 2 & 4 & 1 & 3 & 4 \end{bmatrix},$$

and an array of integer indices indicating where in $w(A)$ a given row's values start,

$$u(A) = \begin{bmatrix} 1 & 3 & 3 & 4 & 7 \end{bmatrix},$$

with a padded value at the end to guarantee that

$$u(A)[n+1] - u(A)[n]$$

is the number of non-zero elements in row n of the matrix (here 2, 0, 1, and 3). Note that because the second row has no non-zero elements both the second and third elements of $u(A)$ correspond to the third element of $w(A)$, which is 52. The values $(w(A), v(A), u(A))$ are sufficient to reconstruct A.

The values are structured so that there is a real value and integer column index for each non-zero entry in the array, plus one integer for each row of the matrix, plus one for padding. There is also underlying storage for internal container pointers and sizes. The total memory usage is roughly $12K + M$ bytes plus a small constant overhead, which is often considerably fewer bytes than the $M \times N$ required to store a dense matrix. Even more importantly, zero values do not introduce derivatives under multiplication or addition, so many storage and evaluation steps are saved when sparse matrices are multiplied.

8.2. Conversion functions

Conversion functions between dense and sparse matrices are provided.

Dense to sparse conversion

Converting a dense matrix m to a sparse representation produces a vector w and two integer arrays, u and v.

`vector` **`csr_extract_w`**`(matrix a)`
Return non-zero values in matrix a; see section compressed row storage.

Available since 2.8

`array[] int` **`csr_extract_v`**`(matrix a)`
Return column indices for values in `csr_extract_w(a)`; see compressed row storage.

Available since 2.8

`array[] int` **`csr_extract_u`**`(matrix a)`
Return array of row starting indices for entries in `csr_extract_w(a)` followed by the size of `csr_extract_w(a)` plus one; see section compressed row storage.

Available since 2.8

`tuple(vector, array[] int, array[] int)` **`csr_extract`**`(matrix a)`
Return all three components of the CSR representation of the matrix `a`; see section compressed row storage. This function is equivalent to `(csr_extract_w(a), csr_extract_v(a), csr_extract_u(a))`.

Available since 2.33

Sparse to dense conversion

To convert a sparse matrix representation to a dense matrix, there is a single function.

`matrix` **`csr_to_dense_matrix`**`(int m, int n, vector w, array[] int v, array[] int u)`
Return dense m × n matrix with non-zero matrix entries w, column indices v, and row starting indices u; the vector w and array v must be the same size (corresponding to the total number of nonzero entries in the matrix), array v must have index values bounded by m, array u must have length equal to m + 1 and contain index values bounded by the number of nonzeros (except for the last entry, which must be equal to the number of nonzeros plus one). See section compressed row storage for more details.

Available since 2.10

8.3. Sparse matrix arithmetic

Sparse matrix multiplication

The only supported operation is the multiplication of a sparse matrix A and a dense vector b to produce a dense vector $A\,b$. Multiplying a dense row vector b and a sparse matrix A can be coded using transposition as

$$b\,A = (A^\top\,b^\top)^\top,$$

but care must be taken to represent $A^\top$ rather than A as a sparse matrix.

`vector` **`csr_matrix_times_vector`**`(int m, int n, vector w, array[] int v, array[] int u, vector b)`
Multiply the m × n matrix represented by values w, column indices v, and row start indices u by the vector b; see compressed row storage.

Available since 2.18

9. Mixed Operations

These functions perform conversions between Stan containers matrix, vector, row vector and arrays.

`matrix` **`to_matrix`**`(matrix m)`
Return the matrix `m` itself.

Available since 2.3

`complex_matrix` **`to_matrix`**`(complex_matrix m)`
Return the matrix `m` itself.

Available since 2.30

`matrix` **`to_matrix`**`(vector v)`
Convert the column vector `v` to a `size(v)` by 1 matrix.

Available since 2.3

`complex_matrix` **`to_matrix`**`(complex_vector v)`
Convert the column vector `v` to a `size(v)` by 1 matrix.

Available since 2.30

`matrix` **`to_matrix`**`(row_vector v)`
Convert the row vector `v` to a 1 by `size(v)` matrix.

Available since 2.3

`complex_matrix` **`to_matrix`**`(complex_row_vector v)`
Convert the row vector `v` to a 1 by `size(v)` matrix.

Available since 2.30

`matrix` **`to_matrix`**`(matrix M, int m, int n)`
Convert a matrix `A` to a matrix with `m` rows and `n` columns filled in column-major order.

Available since 2.15

`complex_matrix` **`to_matrix`**`(complex_matrix M, int m, int n)`
Convert a matrix `A` to a matrix with `m` rows and `n` columns filled in column-major order.

Available since 2.30

`matrix` **`to_matrix`**`(vector v, int m, int n)`
Convert a vector `v` to a matrix with `m` rows and `n` columns filled in column-major order.

Available since 2.15

`complex_matrix` **`to_matrix`**`(complex_vector v, int m, int n)`
Convert a vector `v` to a matrix with `m` rows and `n` columns filled in column-major order.

Available since 2.30

`matrix` **`to_matrix`**`(row_vector v, int m, int n)`
Convert a row_vector `v` to a matrix with `m` rows and `n` columns filled in column-major order.

Available since 2.15

`complex_matrix` **`to_matrix`**`(complex_row_vector v, int m, int n)`
Convert a row vector `v` to a matrix with `m` rows and `n` columns filled in column-major order.

Available since 2.30

`matrix` **`to_matrix`**`(matrix A, int m, int n, int col_major)`
Convert a matrix `A` to a matrix with `m` rows and `n` columns filled in row-major order if `col_major` equals 0 (otherwise, they get filled in column-major order).

Available since 2.15

`complex_matrix` **`to_matrix`**`(complex_matrix A, int m, int n, int col_major)`
Convert a matrix `A` to a matrix with `m` rows and `n` columns filled in row-major order if `col_major` equals 0 (otherwise, they get filled in column-major order).

Available since 2.30

`matrix` **`to_matrix`**`(vector v, int m, int n, int col_major)`
Convert a vector `v` to a matrix with `m` rows and `n` columns filled in row-major order if `col_major` equals 0 (otherwise, they get filled in column-major order).

Available since 2.15

`complex_matrix` **`to_matrix`**`(complex_vector v, int m, int n, int col_major)`

Convert a vector `v` to a matrix with `m` rows and `n` columns filled in row-major order if `col_major` equals 0 (otherwise, they get filled in column-major order).

Available since 2.30

`matrix` **`to_matrix`**`(row_vector v, int m, int n, int col_major)`
Convert a row vector `v` to a matrix with `m` rows and `n` columns filled in row-major order if `col_major` equals 0 (otherwise, they get filled in column-major order).

Available since 2.15

`complex_matrix` **`to_matrix`**`(complex_row_vector v, int m, int n, int col_major)`
Convert a row vector `v` to a matrix with `m` rows and `n` columns filled in row-major order if `col_major` equals 0 (otherwise, they get filled in column-major order).

Available since 2.30

`matrix` **`to_matrix`**`(array[] real a, int m, int n)`
Convert a one-dimensional array `a` to a matrix with `m` rows and `n` columns filled in column-major order.

Available since 2.15

`matrix` **`to_matrix`**`(array[] int a, int m, int n)`
Convert a one-dimensional array `a` to a matrix with `m` rows and `n` columns filled in column-major order.

Available since 2.15

`complex_matrix` **`to_matrix`**`(array[] complex a, int m, int n)`
Convert a one-dimensional array `a` to a matrix with `m` rows and `n` columns filled in column-major order.

Available since 2.30

`matrix` **`to_matrix`**`(array[] real a, int m, int n, int col_major)`
Convert a one-dimensional array `a` to a matrix with `m` rows and `n` columns filled in row-major order if `col_major` equals 0 (otherwise, they get filled in column-major order).

Available since 2.15

`matrix` **`to_matrix`**`(array[] int a, int m, int n, int col_major)`
Convert a one-dimensional array `a` to a matrix with `m` rows and `n` columns filled in

row-major order if `col_major` equals 0 (otherwise, they get filled in column-major order).

Available since 2.15

`complex_matrix` **`to_matrix`**`(array[] complex a, int m, int n, int col_major)`
Convert a one-dimensional array `a` to a matrix with `m` rows and `n` columns filled in row-major order if `col_major` equals 0 (otherwise, they get filled in column-major order).

Available since 2.30

`matrix` **`to_matrix`**`(array[] row_vector vs)`
Convert a one-dimensional array of row vectors to a matrix, where the size of the array is the number of rows of the resulting matrix and the length of row vectors is the number of columns.

Available since 2.28

`complex_matrix` **`to_matrix`**`(array[] complex_row_vector vs)`
Convert a one-dimensional array of row vectors to a matrix, where the size of the array is the number of rows of the resulting matrix and the length of row vectors is the number of columns.

Available since 2.30

`matrix` **`to_matrix`**`(array[,] real a)`
Convert the two dimensional array `a` to a matrix with the same dimensions and indexing order.

Available since 2.3

`matrix` **`to_matrix`**`(array[,] int a)`
Convert the two dimensional array `a` to a matrix with the same dimensions and indexing order. If any of the dimensions of `a` are zero, the result will be a 0×0 matrix.

Available since 2.3

`complex_matrix` **`to_matrix`**`(array[,] complex a )`
Convert the two dimensional array `a` to a matrix with the same dimensions and indexing order.

Available since 2.30

`vector` **`to_vector`**`(matrix m)`
Convert the matrix `m` to a column vector in column-major order.

Available since 2.0

`complex_vector` **`to_vector`**`(complex_matrix m)`
Convert the matrix `m` to a column vector in column-major order.

Available since 2.30

`vector` **`to_vector`**`(vector v)`
Return the column vector `v` itself.

Available since 2.3

`complex_vector` **`to_vector`**`(complex_vector v)`
Return the column vector `v` itself.

Available since 2.30

`vector` **`to_vector`**`(row_vector v)`
Convert the row vector `v` to a column vector.

Available since 2.3

`complex_vector` **`to_vector`**`(complex_row_vector v)`
Convert the row vector `v` to a column vector.

Available since 2.30

`vector` **`to_vector`**`(array[] real a)`
Convert the one-dimensional array `a` to a column vector.

Available since 2.3

`vector` **`to_vector`**`(array[] int a)`
Convert the one-dimensional integer array `a` to a column vector.

Available since 2.3

`complex_vector` **`to_vector`**`(array[] complex a)`
Convert the one-dimensional complex array `a` to a column vector.

Available since 2.30

`row_vector` **`to_row_vector`**`(matrix m)`
Convert the matrix `m` to a row vector in column-major order.

Available since 2.3

`complex_row_vector` **`to_row_vector`**`(complex_matrix m)`
Convert the matrix `m` to a row vector in column-major order.

Available since 2.30

`row_vector` **`to_row_vector`**`(vector v)`
Convert the column vector `v` to a row vector.

Available since 2.3

`complex_row_vector` **`to_row_vector`**`(complex_vector v)`
Convert the column vector `v` to a row vector.

Available since 2.30

`row_vector` **`to_row_vector`**`(row_vector v)`
Return the row vector `v` itself.

Available since 2.3

`complex_row_vector` **`to_row_vector`**`(complex_row_vector v)`
Return the row vector `v` itself.

Available since 2.30

`row_vector` **`to_row_vector`**`(array[] real a)`
Convert the one-dimensional array `a` to a row vector.

Available since 2.3

`row_vector` **`to_row_vector`**`(array[] int a)`
Convert the one-dimensional array `a` to a row vector.

Available since 2.3

`complex_row_vector` **`to_row_vector`**`(array[] complex a)`
Convert the one-dimensional complex array `a` to a row vector.

Available since 2.30

`array[,] real` **`to_array_2d`**`(matrix m)`
Convert the matrix `m` to a two dimensional array with the same dimensions and indexing order.

Available since 2.3

`array[,] complex` **`to_array_2d`**`(complex_matrix m)`
Convert the matrix `m` to a two dimensional array with the same dimensions and

indexing order.

Available since 2.30

`array[] real` **`to_array_1d`**`(vector v)`
Convert the column vector `v` to a one-dimensional array.

Available since 2.3

`array[] complex` **`to_array_1d`**`(complex_vector v)`
Convert the column vector `v` to a one-dimensional array.

Available since 2.30

`array[] real` **`to_array_1d`**`(row_vector v)`
Convert the row vector `v` to a one-dimensional array.

Available since 2.3

`array[] complex` **`to_array_1d`**`(complex_row_vector v)`
Convert the row vector `v` to a one-dimensional array.

Available since 2.30

`array[] real` **`to_array_1d`**`(matrix m)`
Convert the matrix `m` to a one-dimensional array in column-major order.

Available since 2.3

`array[] real` **`to_array_1d`**`(complex_matrix m)`
Convert the matrix `m` to a one-dimensional array in column-major order.

Available since 2.30

`array[] real` **`to_array_1d`**`(array[...] real a)`
Convert the array `a` (of any dimension up to 10) to a one-dimensional array in row-major order.

Available since 2.3

`array[] int` **`to_array_1d`**`(array[...] int a)`
Convert the array `a` (of any dimension up to 10) to a one-dimensional array in row-major order.

Available since 2.3

`array[] complex` **`to_array_1d`**`(array[...] complex a)`
Convert the array `a` (of any dimension up to 10) to a one-dimensional array in

row-major order.

Available since 2.30

10. Compound Arithmetic and Assignment

Compound arithmetic and assignment statements combine an arithmetic operation and assignment, replacing a statement such as

```
x = x op y;
```

with the more compact compound form

```
x op= y;
```

For example, `x = x + 1;` may be replaced with `x += 1;`. This works for all types that support arithmetic, including the scalar types `int`, `real`, `complex`, the real matrix types `vector`, `row_vector`, and `matrix`, and the complex matrix types, `complex_vector`, `complex_row_vector`, and `complex_matrix`.

10.1. Compound addition and assignment

Compound addition and assignment works wherever the corresponding addition and assignment would be well formed.

`void` **`operator+=`**`(T x, U y)`
`x += y` is equivalent to `x = x + y`. Defined for all types `T` and `U` where `T = T + U` is well formed.

Available since 2.17, complex signatures added in 2.30

10.2. Compound subtraction and assignment

Compound addition and assignment works wherever the corresponding subtraction and assignment would be well formed.

`void` **`operator-=`**`(T x, U y)`
`x -= y` is equivalent to `x = x - y`. Defined for all types `T` and `U` where `T = T - U` is well formed.

Available since 2.17, complex signatures added in 2.30

10.3. Compound multiplication and assignment

Compound multiplication and assignment works wherever the corresponding multiplication and assignment would be well formed.

`void` **`operator*=`**`(T x, U y)`
`x *= y` is equivalent to `x = x * y`. Defined for all types `T` and `U` where `T = T * U` is well formed.

Available since 2.17, complex signatures added in 2.30

10.4. Compound division and assignment

Compound division and assignment works wherever the corresponding division and assignment would be well formed.

`void` **`operator/=`**`(T x, U y)`
`x /= y` is equivalent to `x = x / y`. Defined for all types `T` and `U` where `T = T / U` is well formed.

Available since 2.17, complex signatures added in 2.30

10.5. Compound elementwise multiplication and assignment

Compound elementwise multiplication and assignment works wherever the corresponding multiplication and assignment would be well formed.

`void` **`operator.*=`**`(T x, U y)`
`x .*= y` is equivalent to `x = x .* y`. Defined for all types `T` and `U` where `T = T .* U` is well formed.

Available since 2.17, complex signatures added in 2.30

10.6. Compound elementwise division and assignment

Compound elementwise division and assignment works wherever the corresponding division and assignment would be well formed.

`void` **`operator./=`**`(T x, U y)`
`x ./= y` is equivalent to `x = x ./ y`. Defined for all types `T` and `U` where `T = T ./ U` is well formed.

Available since 2.17, complex signatures added in 2.30

11. Higher-Order Functions

Stan provides a few higher-order functions that act on other functions. In all cases, the function arguments to the higher-order functions are defined as functions within the Stan language and passed by name to the higher-order functions.

11.1. Algebraic equation solvers

Stan provides two built-in algebraic equation solvers, respectively based on the Newton method and the Powell "dog leg" hybrid method. Empirically the Newton method is found to be faster and its use is recommended for most problems.

An algebraic solver is a higher-order function, i.e. it takes another function as one of its arguments. Other functions in Stan which share this feature are the differential equation solvers (see section Ordinary Differential Equation (ODE) Solvers and Differential Algebraic Equation (DAE) solver). Ordinary Stan functions do not allow functions as arguments.

Specifying an algebraic equation as a function

An algebraic system is specified as an ordinary function in Stan within the function block. The function must return a `vector` and takes in, as its first argument, the unknowns y we wish to solve for, also passed as a `vector`. This argument is followed by additional arguments as specified by the user; we call such arguments *variadic arguments* and denote them `...`. The signature of the algebraic system is then:

```
vector algebra_system (vector y, ...)
```

There is no type restriction for the variadic arguments and each argument can be passed as data or parameter. However users should use parameter arguments only when necessary and mark data arguments with the keyword `data`. In the below example, the last variadic argument, x, is restricted to being data:

```
  vector algebra_system (vector y, vector theta, data vector x)
```

Distinguishing data and parameter is important for computational reasons. Augmenting the total number of parameters increases the cost of propagating derivatives through the solution to the algebraic equation, and ultimately the computational cost of evaluating the gradients.

Call to the algebraic solver

`vector` **`solve_newton`**`(function algebra_system, vector y_guess, ...)`
Solves the algebraic system, given an initial guess, using Newton's method.

Available since 2.31

`vector` **`solve_newton_tol`**`(function algebra_system, vector y_guess, data real scaling_step, data real f_tol, int max_steps, ...)`
Solves the algebraic system, given an initial guess, using Newton's method with additional control parameters for the solver.

Available since 2.31

`vector` **`solve_powell`**`(function algebra_system, vector y_guess, ...)`
Solves the algebraic system, given an initial guess, using Powell's hybrid method.

Available since 2.31

`vector` **`solve_powell_tol`**`(function algebra_system, vector y_guess, data real rel_tol, data real f_tol, int max_steps, ...)`
Solves the algebraic system, given an initial guess, using Powell's hybrid method with additional control parameters for the solver.

Available since 2.31

Arguments to the algebraic solver

The arguments to the algebraic solvers are as follows:

- *`algebra_system`*: function literal referring to a function specifying the system of algebraic equations with signature `(vector, ...):vector`. The arguments represent (1) unknowns, (2) additional parameter and/or data arguments, and the return value contains the value of the algebraic function, which goes to 0 when we plug in the solution to the algebraic system,
- *`y_guess`*: initial guess for the solution, type `vector`,
- `...`: variadic arguments.

The algebraic solvers admit control parameters. While Stan provides default values, the user should be prepared to adjust the control parameters. The following controls are available:

- *`scaling_step`*: *for the Newton solver only*, the scaled-step stopping tolerance, type `real`, data only. If a Newton step is smaller than the scaling step tolerance, the code breaks, assuming the solver is no longer making significant progress. If set to 0, this constraint is ignored. Default value is 10^{-3}.

- `rel_tol`: *for the Powell solver only*, the relative tolerance, type `real`, data only. The relative tolerance is the estimated relative error of the solver and serves to test if a satisfactory solution has been found. Default value is 10^{-10}.
- `function_tol`: function tolerance for the algebraic solver, type `real`, data only. After convergence of the solver, the proposed solution is plugged into the algebraic system and its norm is compared to the function tolerance. If the norm is below the function tolerance, the solution is deemed acceptable. Default value is 10^{-6}.
- `max_num_steps`: maximum number of steps to take in the algebraic solver, type `int`, data only. If the solver reaches this number of steps, it breaks and returns an error message. Default value is 200.

The difference in which control parameters are available has to do with the underlying implementations for the solvers and the control parameters these implementations support. The Newton solver is based on KINSOL from the SUNDIAL suites, while the Powell solver uses a module from the Eigen library.

Return value

The return value for the algebraic solver is an object of type `vector`, with values which, when plugged in as `y` make the algebraic function go to 0 (approximately, within the specified function tolerance).

Sizes and parallel arrays

Certain sizes have to be consistent. The initial guess, return value of the solver, and return value of the algebraic function must all be the same size.

Algorithmic details

Stan offers two methods to solve algebraic equations. `solve_newton` and `solve_newton_tol` use the Newton method, a first-order derivative based numerical solver. The Stan code builds on the implementation in KINSOL from the SUNDIALS suite (Hindmarsh et al. 2005). For many problems, we find that the Newton method is faster than the Powell method. If however Newton's method performs poorly, either failing to or requiring an excessively long time to converge, the user should be prepared to switch to the Powell method.

`solve_powell` and `solve_powell_tol` are based on the Powell hybrid method (Powell 1970), which also uses first-order derivatives. The Stan code builds on the implementation of the hybrid solver in the unsupported module for nonlinear optimization problems of the Eigen library (Guennebaud, Jacob, et al. 2010). This solver is in turn based on the algorithm developed for the package MINPACK-1 (Jorge J. More 1980).

For both solvers, derivatives are propagated through the solution to the algebraic solution using the implicit function theorem and an adjoint method of automatic differentiation; for a discussion on this topic, see (Gaebler 2021) and (Margossian and Betancourt 2022).

11.2. Ordinary differential equation (ODE) solvers

Stan provides several higher order functions for solving initial value problems specified as Ordinary Differential Equations (ODEs).

Solving an initial value ODE means given a set of differential equations $y'(t, \theta) = f(t, y, \theta)$ and initial conditions $y(t_0, \theta)$, solving for y at a sequence of times $t_0 < t_1 < t_2, \cdots < t_n$. $f(t, y, \theta)$ is referred to here as the ODE system function.

$f(t, y, \theta)$ will be defined as a function with a certain signature and provided along with the initial conditions and output times to one of the ODE solver functions.

To make it easier to write ODEs, the solve functions take extra arguments that are passed along unmodified to the user-supplied system function. Because there can be any number of these arguments and they can be of different types, they are denoted below as `...`. The types of the arguments represented by `...` in the ODE solve function call must match the types of the arguments represented by `...` in the user-supplied system function.

Non-stiff solver

`array[] vector` **`ode_rk45`**`(function ode, vector initial_state, real initial_time, array[] real times, ...)`
Solves the ODE system for the times provided using the Dormand-Prince algorithm, a 4th/5th order Runge-Kutta method.

Available since 2.24

`array[] vector` **`ode_rk45_tol`**`(function ode, vector initial_state, real initial_time, array[] real times, data real rel_tol, data real abs_tol, int max_num_steps, ...)`
Solves the ODE system for the times provided using the Dormand-Prince algorithm, a 4th/5th order Runge-Kutta method with additional control parameters for the solver.

Available since 2.24

`array[] vector` **`ode_ckrk`**`(function ode, vector initial_state, real initial_time, array[] real times, ...)`
Solves the ODE system for the times provided using the Cash-Karp algorithm, a

4th/5th order explicit Runge-Kutta method.

Available since 2.27

`array[] vector` **`ode_ckrk_tol`**`(function ode, vector initial_state, real initial_time, array[] real times, data real rel_tol, data real abs_tol, int max_num_steps, ...)`
Solves the ODE system for the times provided using the Cash-Karp algorithm, a 4th/5th order explicit Runge-Kutta method with additional control parameters for the solver.

Available since 2.27

`array[] vector` **`ode_adams`**`(function ode, vector initial_state, real initial_time, array[] real times, ...)`
Solves the ODE system for the times provided using the Adams-Moulton method.

Available since 2.24

`array[] vector` **`ode_adams_tol`**`(function ode, vector initial_state, real initial_time, array[] real times, data real rel_tol, data real abs_tol, int max_num_steps, ...)`
Solves the ODE system for the times provided using the Adams-Moulton method with additional control parameters for the solver.

Available since 2.24

Stiff solver

`array[] vector` **`ode_bdf`**`(function ode, vector initial_state, real initial_time, array[] real times, ...)`
Solves the ODE system for the times provided using the backward differentiation formula (BDF) method.

Available since 2.24

`array[] vector` **`ode_bdf_tol`**`(function ode, vector initial_state, real initial_time, array[] real times, data real rel_tol, data real abs_tol, int max_num_steps, ...)`
Solves the ODE system for the times provided using the backward differentiation formula (BDF) method with additional control parameters for the solver.

Available since 2.24

Adjoint solver

`array[] vector` **`ode_adjoint_tol_ctl`**`(function ode, vector initial_state, real initial_time, array[] real times, data real rel_tol_forward, data vector abs_tol_forward, data real rel_tol_backward, data vector abs_tol_backward, int max_num_steps, int num_steps_between_checkpoints, int interpolation_polynomial, int solver_forward, int solver_backward, ...)`

Solves the ODE system for the times provided using the adjoint ODE solver method from CVODES. The adjoint ODE solver requires a checkpointed forward in time ODE integration, a backwards in time integration that makes uses of an interpolated version of the forward solution, and the solution of a quadrature problem (the number of which depends on the number of parameters passed to the solve). The tolerances and numeric methods used for the forward solve, backward solve, quadratures, and interpolation can all be configured.

Available since 2.27

ODE system function

The first argument to one of the ODE solvers is always the ODE system function. The ODE system function must have a `vector` return type, and the first two arguments must be a `real` and `vector` in that order. These two arguments are followed by the variadic arguments that are passed through from the ODE solve function call:

```
vector ode(real time, vector state, ...)
```

The ODE system function should return the derivative of the state with respect to time at the time and state provided. The length of the returned vector must match the length of the state input into the function.

The arguments to this function are:

- *`time`*, the time to evaluate the ODE system
- *`state`*, the state of the ODE system at the time specified
- *`...`*, sequence of arguments passed unmodified from the ODE solve function call. The types here must match the types in the `...` arguments of the ODE solve function call.

Arguments to the ODE solvers

The arguments to the ODE solvers in both the stiff and non-stiff solvers are the same. The arguments to the adjoint ODE solver are different; see Arguments to the adjoint ODE solver.

- *ode*: ODE system function,
- *initial_state*: initial state, type `vector`,
- *initial_time*: initial time, type `real`,
- *times*: solution times, type `array[] real`,
- `...`: sequence of arguments that will be passed through unmodified to the ODE system function. The types here must match the types in the `...` arguments of the ODE system function.

For the versions of the ode solver functions ending in `_tol`, these three parameters must be provided after `times` and before the `...` arguments:

- `data` *rel_tol*: relative tolerance for the ODE solver, type `real`, data only,
- `data` *abs_tol*: absolute tolerance for the ODE solver, type `real`, data only, and
- *max_num_steps*: maximum number of steps to take between output times in the ODE solver, type `int`, data only.

Because the tolerances are `data` arguments, they must be defined in either the data or transformed data blocks. They cannot be parameters, transformed parameters or functions of parameters or transformed parameters.

Arguments to the adjoint ODE solver

The arguments to the adjoint ODE solver are different from those for the other functions (for those see Arguments to the ODE solvers).

- *ode*: ODE system function,
- *initial_state*: initial state, type `vector`,
- *initial_time*: initial time, type `real`,
- *times*: solution times, type `array[] real`,
- `data` *rel_tol_forward*: Relative tolerance for forward solve, type `real`, data only,
- `data` *abs_tol_forward*: Absolute tolerance vector for each state for forward solve, type `vector`, data only,

- `data` *`rel_tol_backward`*: Relative tolerance for backward solve, type `real`, data only,
- `data` *`abs_tol_backward`*: Absolute tolerance vector for each state for backward solve, type `vector`, data only,
- `data` *`rel_tol_quadrature`*: Relative tolerance for backward quadrature, type `real`, data only,
- `data` *`abs_tol_quadrature`*: Absolute tolerance for backward quadrature, type `real`, data only,
- `data` *`max_num_steps`*: Maximum number of time-steps to take in integrating the ODE solution between output time points for forward and backward solve, type `int`, data only,
- *`num_steps_between_checkpoints`*: number of steps between checkpointing forward solution, type `int`, data only,
- *`interpolation_polynomial`*: can be 1 for hermite or 2 for polynomial interpolation method of CVODES, type `int`, data only,
- *`solver_forward`*: solver used for forward ODE problem: 1=Adams (non-stiff), 2=BDF (stiff), type `int`, data only,
- *`solver_backward`*: solver used for backward ODE problem: 1=Adams (non-stiff), 2=BDF (stiff), type `int`, data only.
- `...`: sequence of arguments that will be passed through unmodified to the ODE system function. The types here must match the types in the `...` arguments of the ODE system function.

Because the tolerances are `data` arguments, they must be defined in either the data or transformed data blocks. They cannot be parameters, transformed parameters or functions of parameters or transformed parameters.

Return values

The return value for the ODE solvers is an array of vectors (type `array[] vector`), one vector representing the state of the system at every time in specified in the `times` argument.

Array and vector sizes

The sizes must match, and in particular, the following groups are of the same size:

- state variables passed into the system function, derivatives returned by the system function, initial state passed into the solver, and length of each vector in the output,

- number of solution times and number of vectors in the output.

11.3. Differential-Algebraic equation (DAE) solver

Stan provides two higher order functions for solving initial value problems specified as Differential-Algebraic Equations (DAEs) with index-1 (Serban et al. 2021).

Solving an initial value DAE means given a set of residual functions $r(y'(t,\theta), y(t,\theta), t)$ and initial conditions $(y(t_0,\theta), y'(t_0,\theta))$, solving for y at a sequence of times $t_0 < t_1 \leq t_2, \cdots \leq t_n$. The residual function $r(y', y, t, \theta)$ will be defined as a function with a certain signature and provided along with the initial conditions and output times to one of the DAE solver functions.

Similar to ODE solvers, the DAE solver function takes extra arguments that are passed along unmodified to the user-supplied system function. Because there can be any number of these arguments and they can be of different types, they are denoted below as `...`, and the types of these arguments, also represented by `...` in the DAE solver call, must match the types of the arguments represented by `...` in the user-supplied system function.

The DAE solver

`array[] vector` **`dae`**`(function residual, vector initial_state, vector initial_state_derivative, data real initial_time, data array[] real times, ...)`
Solves the DAE system using the backward differentiation formula (BDF) method (Serban et al. 2021).

Available since 2.29

`array[] vector` **`dae_tol`**`(function residual, vector initial_state, vector initial_state_derivative, data real initial_time, data array[] real times, data real rel_tol, data real abs_tol, int max_num_steps, ...)`
Solves the DAE system for the times provided using the backward differentiation formula (BDF) method with additional control parameters for the solver.

Available since 2.29

DAE system function

The first argument to the DAE solver is the DAE residual function. The DAE residual function must have a `vector` return type, and the first three arguments must be a `real`, `vector`, and `vector`, in that order. These three arguments are followed by the variadic arguments that are passed through from the DAE solver function call:

```
vector residual(real time, vector state, vector state_derivative, ...)
```

The DAE residual function should return the residuals at the time and state provided. The length of the returned vector must match the length of the state input into the function.

The arguments to this function are:

- *`time`*, the time to evaluate the DAE system
- *`state`*, the state of the DAE system at the time specified
- *`state_derivative`*, the time derivatives of the state of the DAE system at the time specified
- *`...`*, sequence of arguments passed unmodified from the DAE solve function call. The types here must match the types in the `...` arguments of the DAE solve function call.

Arguments to the DAE solver

The arguments to the DAE solver are

- *`residual`*: DAE residual function,
- *`initial_state`*: initial state, type `vector`,
- *`initial_state_derivative`*: time derivative of the initial state, type `vector`,
- *`initial_time`*: initial time, type `data real`,
- *`times`*: solution times, type `data array[] real`,
- `...`: sequence of arguments that will be passed through unmodified to the DAE residual function. The types here must match the types in the `...` arguments of the DAE residual function.

For `dae_tol`, the following three parameters must be provided after `times` and before the `...` arguments:

- `data` *`rel_tol`*: relative tolerance for the DAE solver, type `real`, data only,
- `data` *`abs_tol`*: absolute tolerance for the DAE solver, type `real`, data only, and

- *max_num_steps*: maximum number of steps to take between output times in the DAE solver, type `int`, data only.

Because the tolerances are `data` arguments, they must be supplied as primitive numerics or defined in either the data or transformed data blocks. They cannot be parameters, transformed parameters or functions of parameters or transformed parameters.

Consistency of the initial conditions

The user is responsible to ensure the residual function becomes zero at the initial time, `t0`, when the arguments `initial_state` and `initial_state_derivative` are introduced as `state` and `state_derivative`, respectively.

Return values

The return value for the DAE solvers is an array of vectors (type `array[] vector`), one vector representing the state of the system at every time specified in the `times` argument.

Array and vector sizes

The sizes must match, and in particular, the following groups are of the same size:

- state variables and state derivatives passed into the residual function, the residual returned by the residual function, initial state and initial state derivatives passed into the solver, and length of each vector in the output,
- number of solution times and number of vectors in the output.

11.4. 1D integrator

Stan provides a built-in mechanism to perform 1D integration of a function via quadrature methods.

It operates similarly to the algebraic solver and the ordinary differential equations solver in that it allows as an argument a function.

Like both of those utilities, some of the arguments are limited to data only expressions. These expressions must not contain variables other than those declared in the data or transformed data blocks.

Specifying an integrand as a function

Performing a 1D integration requires the integrand to be specified somehow. This is done by defining a function in the Stan functions block with the special signature:

```
real integrand(real x, real xc, array[] real theta,
               array[] real x_r, array[] int x_i)
```

The function should return the value of the integrand evaluated at the point x.

The argument of this function are:

- *x*, the independent variable being integrated over
- *xc*, a high precision version of the distance from x to the nearest endpoint in a definite integral (for more into see section Precision Loss).
- *theta*, parameter values used to evaluate the integral
- *x_r*, data values used to evaluate the integral
- *x_i*, integer data used to evaluate the integral

Like algebraic solver and the differential equations solver, the 1D integrator separates parameter values, `theta`, from data values, `x_r`.

Call to the 1D integrator

`real` **`integrate_1d`** `(function integrand, real a, real b, array[] real theta, array[] real x_r, array[] int x_i)`
Integrates the integrand from a to b.

Available since 2.23

`real` **`integrate_1d`** `(function integrand, real a, real b, array[] real theta, array[] real x_r, array[] int x_i, real relative_tolerance)`
Integrates the integrand from a to b with the given relative tolerance.

Available since 2.23

Arguments to the 1D integrator

The arguments to the 1D integrator are as follows:

- *integrand*: function literal referring to a function specifying the integrand with signature `(real, real, array[] real, array[] real, array[] int):real` The arguments represent
  - (1) where integrand is evaluated,
  - (2) distance from evaluation point to integration limit for definite integrals,
  - (3) parameters,
  - (4) real data

    - (5) integer data, and the return value is the integrand evaluated at the given point,
- *a*: left limit of integration, may be negative infinity, type `real`,
- *b*: right limit of integration, may be positive infinity, type `real`,
- *theta*: parameters only, type `array[] real`,
- *x_r*: real data only, type `array[] real`,
- *x_i*: integer data only, type `array[] int`.

A `relative_tolerance` argument can optionally be provided for more control over the algorithm:

- *relative_tolerance*: relative tolerance for the 1d integrator, type `real`, data only.

Return value

The return value for the 1D integrator is a `real`, the value of the integral.

Zero-crossing integrals

For numeric stability, integrals on the (possibly infinite) interval (a, b) that cross zero are split into two integrals, one from $(a, 0)$ and one from $(0, b)$. Each integral is separately integrated to the given `relative_tolerance`.

Precision loss near limits of integration in definite integrals

When integrating certain definite integrals, there can be significant precision loss in evaluating the integrand near the endpoints. This has to do with the breakdown in precision of double precision floating point values when adding or subtracting a small number from a number much larger than it in magnitude (for instance, `1.0 - x`). `xc` (as passed to the integrand) is a high-precision version of the distance between `x` and the definite integral endpoints and can be used to address this issue. More information (and an example where this is useful) is given in the User's Guide. For zero crossing integrals, `xc` will be a high precision version of the distance to the endpoints of the two smaller integrals. For any integral with an endpoint at negative infinity or positive infinity, `xc` is set to `NaN`.

Algorithmic details

Internally the 1D integrator uses the double-exponential methods in the Boost 1D quadrature library. Boost in turn makes use of quadrature methods developed in (Takahasi and Mori 1974), (Mori 1978), (Bailey, Jeyabalan, and Li 2005), and (Tanaka et al. 2009).

The gradients of the integral are computed in accordance with the Leibniz integral rule. Gradients of the integrand are computed internally with Stan's automatic differentiation.

11.5. Reduce-sum function

Stan provides a higher-order reduce function for summation. A function which returns a scalar `g: U -> real` is mapped to every element of a list of type `array[] U`, `{ x1, x2, ... }` and all the results are accumulated,

```
g(x1) + g(x2) + ...
```

For efficiency reasons the reduce function doesn't work with the element-wise evaluated function `g` itself, but instead works through evaluating partial sums, `f: array[] U -> real`, where:

```
f({ x1 }) = g(x1)
f({ x1, x2 }) = g(x1) + g(x2)
f({ x1, x2, ... }) = g(x1) + g(x2) + ...
```

Mathematically the summation reduction is associative and forming arbitrary partial sums in an arbitrary order will not change the result. However, floating point numerics on computers only have a limited precision such that associativity does not hold exactly. This implies that the order of summation determines the exact numerical result. For this reason, the higher-order reduce function is available in two variants:

- `reduce_sum`: Automatically choose partial sums partitioning based on a dynamic scheduling algorithm.
- `reduce_sum_static`: Compute the same sum as `reduce_sum`, but partition the input in the same way for given data set (in `reduce_sum` this partitioning might change depending on computer load). This should result in stable numerical evaluations.

Specifying the reduce-sum function

The higher-order reduce function takes a partial sum function `f`, an array argument `x` (with one array element for each term in the sum), a recommended `grainsize`, and a set of shared arguments. This representation allows parallelization of the resultant sum.

`real` **`reduce_sum`**`(F f, array[] T x, int grainsize, T1 s1, T2 s2, ...)`
`real` **`reduce_sum_static`**`(F f, array[] T x, int grainsize, T1 s1, T2 s2, ...)`

Returns the equivalent of `f(x, 1, size(x), s1, s2, ...)`, but computes the result in parallel by breaking the array `x` into independent partial sums. `s1, s2, ...` are shared between all terms in the sum.

Available since 2.23

- *`f`*: function literal referring to a function specifying the partial sum operation. Refer to the partial sum function.
- *`x`*: array of `T`, one for each term of the reduction, `T` can be any type,
- *`grainsize`*: For `reduce_sum`, `grainsize` is the recommended size of the partial sum (`grainsize = 1` means pick totally automatically). For `reduce_sum_static`, `grainsize` determines the maximum size of the partial sums, type `int`,
- *`s1`*: first (optional) shared argument, type `T1`, where `T1` can be any type
- *`s2`*: second (optional) shared argument, type `T2`, where `T2` can be any type,
- *`...`*: remainder of shared arguments, each of which can be any type.

The partial sum function

The partial sum function must have the following signature where the type `T`, and the types of all the shared arguments (`T1`, `T2`, `...`) match those of the original `reduce_sum` (`reduce_sum_static`) call.

```
(array[] T x_subset, int start, int end, T1 s1, T2 s2, ...):real
```

The partial sum function returns the sum of the `start` to `end` terms (inclusive) of the overall calculations. The arguments to the partial sum function are:

- *`x_subset`*, the subset of `x` a given partial sum is responsible for computing, type `array[] T`, where `T` matches the type of `x` in `reduce_sum` (`reduce_sum_static`)

- *`start`*, the index of the first term of the partial sum, type `int`

- *`end`*, the index of the last term of the partial sum (inclusive), type `int`

- *`s1`*, first shared argument, type `T1`, matching type of `s1` in `reduce_sum` (`reduce_sum_static`)

- *`s2`*, second shared argument, type `T2`, matching type of `s2` in `reduce_sum` (`reduce_sum_static`)

- *`...`*, remainder of shared arguments, with types matching those in `reduce_sum` (`reduce_sum_static`)

11.6. Map-rect function

Stan provides a higher-order map function. This allows map-reduce functionality to be coded in Stan as described in the user's guide.

Specifying the mapped function

The function being mapped must have a signature identical to that of the function `f` in the following declaration.

```
vector f(vector phi, vector theta,
         data array[] real x_r, data array[] int x_i);
```

The map function returns the sequence of results for the particular shard being evaluated. The arguments to the mapped function are:

- *`phi`*, the sequence of parameters shared across shards
- *`theta`*, the sequence of parameters specific to this shard
- *`x_r`*, sequence of real-valued data
- *`x_i`*, sequence of integer data

All input for the mapped function must be packed into these sequences and all output from the mapped function must be packed into a single vector. The vector of output from each mapped function is concatenated into the final result.

Rectangular map

The rectangular map function operates on rectangular (not ragged) data structures, with parallel data structures for job-specific parameters, job-specific real data, and job-specific integer data.

`vector` **`map_rect`**`(F f, vector phi, array[] vector theta, data array[,] real x_r, data array[,] int x_i)`
Return the concatenation of the results of applying the function f, of type `(vector, vector, array[] real, array[] int):vector` elementwise, i.e., `f(phi, theta[n], x_r[n], x_i[n])` for each `n` in `1:N`, where `N` is the size of the parallel arrays of job-specific/local parameters `theta`, real data `x_r`, and integer data `x_r`. The shared/global parameters `phi` are passed to each invocation of `f`.

Available since 2.18

12. Deprecated Functions

This appendix lists currently deprecated functionality along with how to replace it.

Starting in Stan 2.29, deprecated functions with drop in replacements (such as the renaming of `get_lp` or `multiply_log`) will be removed 3 versions later e.g., functions deprecated in Stan 2.20 will be removed in Stan 2.23 and placed in Removed Functions. The Stan compiler can automatically update these on the behalf of the user for the entire deprecation window and at least one version following the removal.

12.1. Integer division with `operator/`

Deprecated: Using `/` with two integer arguments is interpreted as integer floor division, such that

$$1/2 = 0$$

This is deprecated due to its confusion with real-valued division, where

$$1.0/2.0 = 0.5$$

Replacement: Use the integer division operator `operator%/%` instead.

12.2. integrate_ode_rk45, integrate_ode_adams, integrate_ode_bdf ODE Integrators

These ODE integrator functions have been replaced by those described in Ordinary Differential Equation (ODE) Solvers.

Specifying an ordinary differential equation as a function

A system of ODEs is specified as an ordinary function in Stan within the functions block. The ODE system function must have this function signature:

```
array[] real ode(real time, array[] real state, array[] real theta,
                 array[] real x_r, array[] int x_i);
```

The ODE system function should return the derivative of the state with respect to time at the time provided. The length of the returned real array must match the length of the state input into the function.

The arguments to this function are:

- *time*, the time to evaluate the ODE system
- *state*, the state of the ODE system at the time specified
- *theta*, parameter values used to evaluate the ODE system
- *x_r*, data values used to evaluate the ODE system
- *x_i*, integer data values used to evaluate the ODE system.

The ODE system function separates parameter values, *theta*, from data values, *x_r*, for efficiency in computing the gradients of the ODE.

Non-stiff solver

`array[,] real` **`integrate_ode_rk45`**`(function ode, array[] real initial_state, real initial_time, array[] real times, array[] real theta, array[] real x_r, array[] int x_i)`
Solves the ODE system for the times provided using the Dormand-Prince algorithm, a 4th/5th order Runge-Kutta method.

Available since 2.10, deprecated in 2.24

`array[,] real` **`integrate_ode_rk45`**`(function ode, array[] real initial_state, real initial_time, array[] real times, array[] real theta, array[] real x_r, array[] int x_i, real rel_tol, real abs_tol, int max_num_steps)`
Solves the ODE system for the times provided using the Dormand-Prince algorithm, a 4th/5th order Runge-Kutta method with additional control parameters for the solver.

Available since 2.10, deprecated in 2.24

`array[,] real` **`integrate_ode`**`(function ode, array[] real initial_state, real initial_time, array[] real times, array[] real theta, array[] real x_r, array[] int x_i)`
Solves the ODE system for the times provided using the Dormand-Prince algorithm, a 4th/5th order Runge-Kutta method.

Available since 2.10, deprecated in 2.24

`array[,] real` **`integrate_ode_adams`**`(function ode, array[] real initial_state, real initial_time, array[] real times, array[] real theta, data array[] real x_r, data array[] int x_i)`
Solves the ODE system for the times provided using the Adams-Moulton method.

Available since 2.23, deprecated in 2.24

`array[,] real` **`integrate_ode_adams`**`(function ode, array[] real initial_state, real initial_time, array[] real times, array[] real theta, data array[] real x_r, data array[] int x_i, data real rel_tol, data real abs_tol, data int max_num_steps)`
Solves the ODE system for the times provided using the Adams-Moulton method with additional control parameters for the solver.

Available since 2.23, deprecated in 2.24

Stiff solver

`array[,] real` **`integrate_ode_bdf`**`(function ode, array[] real initial_state, real initial_time, array[] real times, array[] real theta, data array[] real x_r, data array[] int x_i)`
Solves the ODE system for the times provided using the backward differentiation formula (BDF) method.

Available since 2.10, deprecated in 2.24

`array[,] real` **`integrate_ode_bdf`**`(function ode, array[] real initial_state, real initial_time, array[] real times, array[] real theta, data array[] real x_r, data array[] int x_i, data real rel_tol, data real abs_tol, data int max_num_steps)`
Solves the ODE system for the times provided using the backward differentiation formula (BDF) method with additional control parameters for the solver.

Available since 2.10, deprecated in 2.24

Arguments to the ODE solvers

The arguments to the ODE solvers in both the stiff and non-stiff cases are as follows.

- *ode*: function literal referring to a function specifying the system of differential equations with signature:

```
(real, array[] real, array[] real, data array[] real, data array[] int):array[] re
```

The arguments represent (1) time, (2) system state, (3) parameters, (4) real data, and (5) integer data, and the return value contains the derivatives with respect to time of the state,

- `initial_state`: initial state, type `array[] real`,
- `initial_time`: initial time, type `int` or `real`,
- `times`: solution times, type `array[] real`,
- `theta`: parameters, type `array[] real`,
- `data x_r`: real data, type `array[] real`, data only, and
- `data x_i`: integer data, type `array[] int`, data only.

For more fine-grained control of the ODE solvers, these parameters can also be provided:

- `data rel_tol`: relative tolerance for the ODE solver, type `real`, data only,
- `data abs_tol`: absolute tolerance for the ODE solver, type `real`, data only, and
- `data max_num_steps`: maximum number of steps to take in the ODE solver, type `int`, data only.

Return values

The return value for the ODE solvers is an array of type `array[,] real`, with values consisting of solutions at the specified times.

Sizes and parallel arrays

The sizes must match, and in particular, the following groups are of the same size:

- state variables passed into the system function, derivatives returned by the system function, initial state passed into the solver, and rows of the return value of the solver,
- solution times and number of rows of the return value of the solver,
- parameters, real data and integer data passed to the solver will be passed to the system function

12.3. algebra_solver, algebra_solver_newton algebraic solvers

These algebraic solver functions have been replaced by those described in Algebraic Equation Solvers..

Specifying an algebraic equation as a function

An algebraic system is specified as an ordinary function in Stan within the function block. The algebraic system function must have this signature:

```
vector algebra_system(vector y, vector theta,
                              data array[] real x_r, array[] int x_i)
```

The algebraic system function should return the value of the algebraic function which goes to 0, when we plug in the solution to the algebraic system.

The argument of this function are:

- *y*, the unknowns we wish to solve for
- *theta*, parameter values used to evaluate the algebraic system
- *x_r*, data values used to evaluate the algebraic system
- *x_i*, integer data used to evaluate the algebraic system

The algebraic system function separates parameter values, *theta*, from data values, *x_r*, for efficiency in propagating the derivatives through the algebraic system.

Call to the algebraic solver

`vector` **`algebra_solver`**`(function algebra_system, vector y_guess, vector theta, data array[] real x_r, array[] int x_i)`
Solves the algebraic system, given an initial guess, using the Powell hybrid algorithm.

Available since 2.17, deprecated in 2.31

`vector` **`algebra_solver`**`(function algebra_system, vector y_guess, vector theta, data array[] real x_r, array[] int x_i, data real rel_tol, data real f_tol, int max_steps)`
Solves the algebraic system, given an initial guess, using the Powell hybrid algorithm with additional control parameters for the solver.

Available since 2.17, deprecated in 2.31

Note: In future releases, the function `algebra_solver` will be deprecated and replaced with `algebra_solver_powell`.

`vector` **`algebra_solver_newton`**`(function algebra_system, vector y_guess, vector theta, data array[] real x_r, array[] int x_i)`
Solves the algebraic system, given an initial guess, using Newton's method.

Available since 2.24, deprecated in 2.31

`vector` **`algebra_solver_newton`**`(function algebra_system, vector y_guess, vector theta, data array[] real x_r, array[] int x_i,`

`data real rel_tol, data real f_tol, int max_steps)`
Solves the algebraic system, given an initial guess, using Newton's method with additional control parameters for the solver.

Available since 2.24, deprecated in 2.31

Arguments to the algebraic solver

The arguments to the algebraic solvers are as follows:

- *`algebra_system`*: function literal referring to a function specifying the system of algebraic equations with signature `(vector, vector, array[] real, array[] int):vector`. The arguments represent (1) unknowns, (2) parameters, (3) real data, and (4) integer data, and the return value contains the value of the algebraic function, which goes to 0 when we plug in the solution to the algebraic system,
- *`y_guess`*: initial guess for the solution, type `vector`,
- *`theta`*: parameters only, type `vector`,
- *`x_r`*: real data only, type `array[] real`, and
- *`x_i`*: integer data only, type `array[] int`.

For more fine-grained control of the algebraic solver, these parameters can also be provided:

- *`rel_tol`*: relative tolerance for the algebraic solver, type `real`, data only,
- *`function_tol`*: function tolerance for the algebraic solver, type `real`, data only,
- *`max_num_steps`*: maximum number of steps to take in the algebraic solver, type `int`, data only.

Return value

The return value for the algebraic solver is an object of type `vector`, with values which, when plugged in as `y` make the algebraic function go to 0.

Sizes and parallel arrays

Certain sizes have to be consistent. The initial guess, return value of the solver, and return value of the algebraic function must all be the same size.

The parameters, real data, and integer data will be passed from the solver directly to the system function.

13. Removed Functions

Functions which once existed in the Stan language and have since been replaced or removed are listed here.

13.1. `multiply_log` and `binomial_coefficient_log` functions

Removed: Currently two non-conforming functions ending in suffix `_log`.

Replacement: Replace `multiply_log(...)` with `lmultiply(...)`. Replace `binomial_coefficient_log(...)` with `lchoose(...)`.

Removed In: Stan 2.33

13.2. `get_lp()` function

Removed: The built-in no-argument function `get_lp()` is deprecated.

Replacement: Use the no-argument function `target()` instead.

Removed In: Stan 2.33

13.3. `fabs` function

Removed: The unary function `fabs` is deprecated.

Replacement: Use the unary function `abs` instead. Note that the return type for `abs` is different for integer overloads, but this replacement is safe due to Stan's type promotion rules.

Removed In: Stan 2.33

13.4. Exponentiated quadratic covariance functions

These covariance functions have been replaced by those described in Gaussian Process Covariance Functions

With magnitude α and length scale l, the exponentiated quadratic kernel is:

$$k(x_i, x_j) = \alpha^2 \exp\left(-\frac{1}{2\rho^2} \sum_{d=1}^{D} (x_{i,d} - x_{j,d})^2\right)$$

`matrix` **`cov_exp_quad`**`(row_vectors x, real alpha, real rho)`
The covariance matrix with an exponentiated quadratic kernel of x.

Available since 2.16, deprecated since 2.20, removed in in 2.33

`matrix` **`cov_exp_quad`**`(vectors x, real alpha, real rho)`
The covariance matrix with an exponentiated quadratic kernel of x.

Available since 2.16, deprecated since 2.20, removed in in 2.33

`matrix` **`cov_exp_quad`**`(array[] real x, real alpha, real rho)`
The covariance matrix with an exponentiated quadratic kernel of x.

Available since 2.16, deprecated since 2.20, removed in in 2.33

`matrix` **`cov_exp_quad`**`(row_vectors x1, row_vectors x2, real alpha, real rho)`
The covariance matrix with an exponentiated quadratic kernel of x1 and x2.

Available since 2.18, deprecated since 2.20, removed in in 2.33

`matrix` **`cov_exp_quad`**`(vectors x1, vectors x2, real alpha, real rho)`
The covariance matrix with an exponentiated quadratic kernel of x1 and x2.

Available since 2.18, deprecated since 2.20, removed in in 2.33

`matrix` **`cov_exp_quad`**`(array[] real x1, array[] real x2, real alpha, real rho)`
The covariance matrix with an exponentiated quadratic kernel of x1 and x2.

Available since 2.18, deprecated since 2.20, removed in in 2.33

13.5. Real arguments to logical operators `operator&&`, `operator||`, and `operator!`

Removed: A nonzero real number (even NaN) was interpreted as true and a zero was interpreted as false.

Replacement: Explicit `x != 0` comparison is preferred instead.

Removed In: Stan 2.34

14. Conventions for Probability Functions

Functions associated with distributions are set up to follow the same naming conventions for both built-in distributions and for user-defined distributions.

14.1. Suffix marks type of function

The suffix is determined by the type of function according to the following table.

function	outcome	suffix
log probability mass function	discrete	`_lpmf`
log probability density function	continuous	`_lpdf`
log cumulative distribution function	any	`_lcdf`
log complementary cumulative distribution function	any	`_lccdf`
random number generator	any	`_rng`

For example, `normal_lpdf` is the log of the normal probability density function (pdf) and `bernoulli_lpmf` is the log of the bernoulli probability mass function (pmf). The log of the corresponding cumulative distribution functions (cdf) use the same suffix, `normal_lcdf` and `bernoulli_lcdf`.

14.2. Argument order and the vertical bar

Each probability function has a specific outcome value and a number of parameters. Following conditional probability notation, probability density and mass functions use a vertical bar to separate the outcome from the parameters of the distribution. For example, `normal_lpdf(y | mu, sigma)` returns the value of mathematical formula $\log \text{Normal}(y \mid \mu, \sigma)$. Cumulative distribution functions separate the outcome from the parameters in the same way (e.g., `normal_lcdf(y_low | mu, sigma)`

14.3. Sampling notation

The notation

```
y ~ normal(mu, sigma);
```

provides the same (proportional) contribution to the model log density as the explicit target density increment,

```
target += normal_lpdf(y | mu, sigma);
```

In both cases, the effect is to add terms to the target log density. The only difference is that the example with the sampling (~) notation drops all additive constants in the log density; the constants are not necessary for any of Stan's sampling, approximation, or optimization algorithms.

14.4. Finite inputs

All of the distribution functions are configured to throw exceptions (effectively rejecting samples or optimization steps) when they are supplied with non-finite arguments. The two cases of non-finite arguments are the infinite values and not-a-number value—these are standard in floating-point arithmetic.

14.5. Boundary conditions

Many distributions are defined with support or constraints on parameters forming an open interval. For example, the normal density function accepts a scale parameter $\sigma > 0$. If $\sigma = 0$, the probability function will throw an exception.

This is true even for (complementary) cumulative distribution functions, which will throw exceptions when given input that is out of the support.

14.6. Pseudorandom number generators

For most of the probability functions, there is a matching pseudorandom number generator (PRNG) with the suffix `_rng`. For example, the function `normal_rng(real, real)` accepts two real arguments, an unconstrained location μ and positive scale $\sigma > 0$, and returns an unconstrained pseudorandom value drawn from $\text{Normal}(\mu, \sigma)$. There are also vectorized forms of random number generators which return more than one random variate at a time.

Restricted to transformed data and generated quantities

Unlike regular functions, the PRNG functions may only be used in the transformed data or generated quantities blocks.

Limited vectorization

Unlike the probability functions, only some of the PRNG functions are vectorized.

14.7. Cumulative distribution functions

For most of the univariate probability functions, there is a corresponding cumulative distribution function, log cumulative distribution function, and log complementary cumulative distribution function.

For a univariate random variable Y with probability function $p_Y(y \mid \theta)$, the cumulative distribution function (CDF) F_Y is defined by

$$F_Y(y) \;=\; \Pr[Y \leq y] \;=\; \int_{-\infty}^{y} p(y \mid \theta)\, \mathrm{d}y.$$

The complementary cumulative distribution function (CCDF) is defined as

$$\Pr[Y > y] \;=\; 1 - F_Y(y).$$

The reason to use CCDFs instead of CDFs in floating-point arithmetic is that it is possible to represent numbers very close to 0 (the closest you can get is roughly 10^{-300}), but not numbers very close to 1 (the closest you can get is roughly $1 - 10^{-15}$).

In Stan, there is a cumulative distribution function for each probability function. For instance, `normal_cdf(y | mu, sigma)` is defined by

$$\int_{-\infty}^{y} \mathrm{Normal}(y \mid \mu, \sigma)\, \mathrm{d}y.$$

There are also log forms of the CDF and CCDF for most univariate distributions. For example, `normal_lcdf(y | mu, sigma)` is defined by

$$\log\left(\int_{-\infty}^{y} \mathrm{Normal}(y \mid \mu, \sigma)\, \mathrm{d}y\right)$$

and `normal_lccdf(y | mu, sigma)` is defined by

$$\log\left(1 - \int_{-\infty}^{y} \mathrm{Normal}(y \mid \mu, \sigma)\, \mathrm{d}y\right).$$

14.8. Vectorization

Stan's univariate log probability functions, including the log density functions, log mass functions, log CDFs, and log CCDFs, all support vectorized function application, with results defined to be the sum of the elementwise application of the function. Some of the PRNG functions support vectorization, see section vectorized PRNG functions for more details.

In all cases, matrix operations are at least as fast and usually faster than loops and vectorized log probability functions are faster than their equivalent form defined with loops. This isn't because loops are slow in Stan, but because more efficient automatic differentiation can be used. The efficiency comes from the fact that a vectorized log probability function only introduces one new node into the expression graph, thus reducing the number of virtual function calls required to compute gradients in C++, as well as from allowing caching of repeated computations.

Stan also overloads the multivariate normal distribution, including the Cholesky-factor form, allowing arrays of row vectors or vectors for the variate and location parameter. This is a huge savings in speed because the work required to solve the linear system for the covariance matrix is only done once.

Stan also overloads some scalar functions, such as `log` and `exp`, to apply to vectors (arrays) and return vectors (arrays). These vectorizations are defined elementwise and unlike the probability functions, provide only minimal efficiency speedups over repeated application and assignment in a loop.

Vectorized function signatures

Vectorized scalar arguments

The normal probability function is specified with the signature

```
normal_lpdf(reals | reals, reals);
```

The pseudotype `reals` is used to indicate that an argument position may be vectorized. Argument positions declared as `reals` may be filled with a real, a one-dimensional array, a vector, or a row-vector. If there is more than one array or vector argument, their types can be anything but their size must match. For instance, it is legal to use `normal_lpdf(row_vector | vector, real)` as long as the vector and row vector have the same size.

Vectorized vector and row vector arguments

The multivariate normal distribution accepting vector or array of vector arguments is written as

```
multi_normal_lpdf(vectors | vectors, matrix);
```

These arguments may be row vectors, column vectors, or arrays of row vectors or column vectors.

Vectorized integer arguments

The pseudotype `ints` is used for vectorized integer arguments. Where it appears either an integer or array of integers may be used.

Evaluating vectorized log probability functions

The result of a vectorized log probability function is equivalent to the sum of the evaluations on each element. Any non-vector argument, namely `real` or `int`, is repeated. For instance, if `y` is a vector of size `N`, `mu` is a vector of size `N`, and `sigma` is a scalar, then

```
ll = normal_lpdf(y | mu, sigma);
```

is just a more efficient way to write

```
ll = 0;
for (n in 1:N) {
  ll = ll + normal_lpdf(y[n] | mu[n], sigma);
}
```

With the same arguments, the vectorized sampling statement

```
y ~ normal(mu, sigma);
```

has the same effect on the total log probability as

```
for (n in 1:N) {
  y[n] ~ normal(mu[n], sigma);
}
```

Evaluating vectorized PRNG functions

Some PRNG functions accept sequences as well as scalars as arguments. Such functions are indicated by argument pseudotypes `reals` or `ints`. In cases of sequence arguments, the output will also be a sequence. For example, the following is allowed in the transformed data and generated quantities blocks.

```
vector[3] mu = // ...
array[3] real x = normal_rng(mu, 3);
```

Argument types

In the case of PRNG functions, arguments marked `ints` may be integers or integer arrays, whereas arguments marked `reals` may be integers or reals, integer or real arrays, vectors, or row vectors.

pseudotype	allowable PRNG arguments
`ints`	`int`, `array[] int`
`reals`	`int`, `array[] int`, `real`, `array[] real`, `vector`, `row_vector`

Dimension matching

In general, if there are multiple non-scalar arguments, they must all have the same dimensions, but need not have the same type. For example, the `normal_rng` function may be called with one vector argument and one real array argument as long as they have the same number of elements.

```
vector[3] mu = // ...
array[3] real sigma = // ...
array[3] real x = normal_rng(mu, sigma);
```

Return type

The result of a vectorized PRNG function depends on the size of the arguments and the distribution's support. If all arguments are scalars, then the return type is a scalar. For a continuous distribution, if there are any non-scalar arguments, the return type is a real array (`array[] real`) matching the size of any of the non-scalar arguments, as all non-scalar arguments must have matching size. Discrete distributions return `ints` and continuous distributions return `reals`, each of appropriate size. The symbol `R` denotes such a return type.

Part II

Discrete Distributions

15. Binary Distributions

Binary probability distributions have support on $\{0,1\}$, where 1 represents the value true and 0 the value false.

15.1. Bernoulli distribution

Probability mass function

If $\theta \in [0,1]$, then for $y \in \{0,1\}$,

$$\text{Bernoulli}(y \mid \theta) = \begin{cases} \theta & \text{if } y = 1\text{, and} \\ 1-\theta & \text{if } y = 0. \end{cases}$$

Distribution statement

`y ~ bernoulli(theta)`

Increment target log probability density with `bernoulli_lupmf(y | theta)`.

Available since 2.0

Stan Functions

`real` **`bernoulli_lpmf`**`(ints y | reals theta)`
The log Bernoulli probability mass of y given chance of success `theta`

Available since 2.12

`real` **`bernoulli_lupmf`**`(ints y | reals theta)`
The log Bernoulli probability mass of y given chance of success theta dropping constant additive terms

Available since 2.25

`real` **`bernoulli_cdf`**`(ints y | reals theta)`
The Bernoulli cumulative distribution function of y given chance of success `theta`

Available since 2.0

`real` **`bernoulli_lcdf`**`(ints y | reals theta)`
The log of the Bernoulli cumulative distribution function of y given chance of success `theta`

Available since 2.12

`real` **`bernoulli_lccdf`**`(ints y | reals theta)`
The log of the Bernoulli complementary cumulative distribution function of y given chance of success `theta`

Available since 2.12

`ints` **`bernoulli_rng`**`(reals theta)`
Generate a Bernoulli variate with chance of success `theta` or an array of Bernoulli variates given an array of `thetas` of the same dimensions; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.18

15.2. Bernoulli distribution, logit parameterization

Stan also supplies a direct parameterization in terms of a logit-transformed chance-of-success parameter. This parameterization is more numerically stable if the chance-of-success parameter is on the logit scale, as with the linear predictor in a logistic regression.

Probability mass function

If $\alpha \in \mathbb{R}$, then for $y \in \{0, 1\}$,

$$\text{BernoulliLogit}(y \mid \alpha) = \text{Bernoulli}(y|\text{logit}^{-1}(\alpha)) = \begin{cases} \text{logit}^{-1}(\alpha) & \text{if } y = 1, \text{ and} \\ 1 - \text{logit}^{-1}(\alpha) & \text{if } y = 0. \end{cases}$$

Distribution statement

`y ~` **`bernoulli_logit`**`(alpha)`

Increment target log probability density with `bernoulli_logit_lupmf(y | alpha)`.

Available since 2.0

Stan Functions

`real` **`bernoulli_logit_lpmf`**`(ints y | reals alpha)`
The log Bernoulli probability mass of y given chance of success `inv_logit(alpha)`

Available since 2.12

`real` **`bernoulli_logit_lupmf`**`(ints y | reals alpha)`
The log Bernoulli probability mass of y given chance of success `inv_logit(alpha)` dropping constant additive terms

Available since 2.25

`R` **`bernoulli_logit_rng`**`(reals alpha)`
Generate a Bernoulli variate with chance of success $\text{logit}^{-1}(\alpha)$; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.18

15.3. Bernoulli-logit generalized linear model (Logistic Regression)

Stan also supplies a single function for a generalized linear model with Bernoulli distribution and logit link function, i.e. a function for a logistic regression. This provides a more efficient implementation of logistic regression than a manually written regression in terms of a Bernoulli distribution and matrix multiplication.

Probability mass function

If $x \in \mathbb{R}^{n \cdot m}, \alpha \in \mathbb{R}^n, \beta \in \mathbb{R}^m$, then for $y \in \{0, 1\}^n$,

$$\text{BernoulliLogitGLM}(y \mid x, \alpha, \beta) = \prod_{1 \le i \le n} \text{Bernoulli}(y_i \mid \text{logit}^{-1}(\alpha_i + x_i \cdot \beta))$$

$$= \prod_{1 \le i \le n} \begin{cases} \text{logit}^{-1}(\alpha_i + \sum_{1 \le j \le m} x_{ij} \cdot \beta_j) & \text{if } y_i = 1, \text{ and} \\ 1 - \text{logit}^{-1}(\alpha_i + \sum_{1 \le j \le m} x_{ij} \cdot \beta_j) & \text{if } y_i = 0. \end{cases}$$

Distribution statement

`y ~` **`bernoulli_logit_glm`**`(x, alpha, beta)`

Increment target log probability density with `bernoulli_logit_glm_lupmf(y | x, alpha, beta)`.

Available since 2.25

Stan Functions

`real` **`bernoulli_logit_glm_lpmf`**`(int y | matrix x, real alpha, vector beta)`
The log Bernoulli probability mass of y given chance of success `inv_logit(alpha + x * beta)`.

Available since 2.23

`real` **`bernoulli_logit_glm_lupmf`**`(int y | matrix x, real alpha, vector beta)`

The log Bernoulli probability mass of y given chance of success `inv_logit(alpha + x * beta)` dropping constant additive terms.

Available since 2.25

`real` **`bernoulli_logit_glm_lpmf`**`(int y | matrix x, vector alpha, vector beta)`
The log Bernoulli probability mass of y given chance of success `inv_logit(alpha + x * beta)`.

Available since 2.23

`real` **`bernoulli_logit_glm_lupmf`**`(int y | matrix x, vector alpha, vector beta)`
The log Bernoulli probability mass of y given chance of success `inv_logit(alpha + x * beta)` dropping constant additive terms.

Available since 2.25

`real` **`bernoulli_logit_glm_lpmf`**`(array[] int y | row_vector x, real alpha, vector beta)`
The log Bernoulli probability mass of y given chance of success `inv_logit(alpha + x * beta)`.

Available since 2.23

`real` **`bernoulli_logit_glm_lupmf`**`(array[] int y | row_vector x, real alpha, vector beta)`
The log Bernoulli probability mass of y given chance of success `inv_logit(alpha + x * beta)` dropping constant additive terms.

Available since 2.25

`real` **`bernoulli_logit_glm_lpmf`**`(array[] int y | row_vector x, vector alpha, vector beta)`
The log Bernoulli probability mass of y given chance of success `inv_logit(alpha + x * beta)`.

Available since 2.23

`real` **`bernoulli_logit_glm_lupmf`**`(array[] int y | row_vector x, vector alpha, vector beta)`
The log Bernoulli probability mass of y given chance of success `inv_logit(alpha + x * beta)` dropping constant additive terms.

Available since 2.25

`real` **`bernoulli_logit_glm_lpmf`**`(array[] int y | matrix x, real alpha, vector beta)`
The log Bernoulli probability mass of y given chance of success `inv_logit(alpha + x * beta)`.

Available since 2.18

`real` **`bernoulli_logit_glm_lupmf`**`(array[] int y | matrix x, real alpha, vector beta)`
The log Bernoulli probability mass of y given chance of success `inv_logit(alpha + x * beta)` dropping constant additive terms.

Available since 2.25

`real` **`bernoulli_logit_glm_lpmf`**`(array[] int y | matrix x, vector alpha, vector beta)`
The log Bernoulli probability mass of y given chance of success `inv_logit(alpha + x * beta)`.

Available since 2.18

`real` **`bernoulli_logit_glm_lupmf`**`(array[] int y | matrix x, vector alpha, vector beta)`
The log Bernoulli probability mass of y given chance of success `inv_logit(alpha + x * beta)` dropping constant additive terms.

Available since 2.25

`array[] int` **`bernoulli_logit_glm_rng`**`(matrix x, vector alpha, vector beta)`
Generate an array of Bernoulli variates with chances of success `inv_logit(alpha + x * beta)`; may only be used in transformed data and generated quantities blocks.

Available since 2.29

`array[] int` **`bernoulli_logit_glm_rng`**`(row_vector x, vector alpha, vector beta)`
Generate an array of Bernoulli variates with chances of success `inv_logit(alpha + x * beta)`; may only be used in transformed data and generated quantities blocks.

Available since 2.29

16. Bounded Discrete Distributions

Bounded discrete probability functions have support on $\{0, \ldots, N\}$ for some upper bound N.

16.1. Binomial distribution

Probability mass function

Suppose $N \in \mathbb{N}$ and $\theta \in [0,1]$, and $n \in \{0, \ldots, N\}$.

$$\text{Binomial}(n \mid N, \theta) = \binom{N}{n} \theta^n (1-\theta)^{N-n}.$$

Log probability mass function

$$\begin{aligned} \log \text{Binomial}(n \mid N, \theta) &= \log \Gamma(N+1) - \log \Gamma(n+1) - \log \Gamma(N-n+1) \\ &\quad + n \log \theta + (N-n) \log(1-\theta), \end{aligned}$$

Gradient of log probability mass function

$$\frac{\partial}{\partial \theta} \log \text{Binomial}(n \mid N, \theta) = \frac{n}{\theta} - \frac{N-n}{1-\theta}$$

Distribution statement

`n ~ binomial(N, theta)`

Increment target log probability density with `binomial_lupmf(n | N, theta)`.

Available since 2.0

Stan functions

`real binomial_lpmf(ints n | ints N, reals theta)`
The log binomial probability mass of n successes in N trials given chance of success theta

Available since 2.12

`real` **`binomial_lupmf`**`(ints n | ints N, reals theta)`
The log binomial probability mass of n successes in N trials given chance of success theta dropping constant additive terms

Available since 2.25

`real` **`binomial_cdf`**`(ints n | ints N, reals theta)`
The binomial cumulative distribution function of n successes in N trials given chance of success theta

Available since 2.0

`real` **`binomial_lcdf`**`(ints n | ints N, reals theta)`
The log of the binomial cumulative distribution function of n successes in N trials given chance of success theta

Available since 2.12

`real` **`binomial_lccdf`**`(ints n | ints N, reals theta)`
The log of the binomial complementary cumulative distribution function of n successes in N trials given chance of success theta

Available since 2.12

`R` **`binomial_rng`**`(ints N, reals theta)`
Generate a binomial variate with N trials and chance of success theta; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.18

16.2. Binomial distribution, logit parameterization

Stan also provides a version of the binomial probability mass function distribution with the chance of success parameterized on the unconstrained logistic scale.

Probability mass function

Suppose $N \in \mathbb{N}$, $\alpha \in \mathbb{R}$, and $n \in \{0, \ldots, N\}$. Then

$$\begin{aligned} \text{BinomialLogit}(n \mid N, \alpha) &= \text{Binomial}(n \mid N, \text{logit}^{-1}(\alpha)) \\ &= \binom{N}{n} \left(\text{logit}^{-1}(\alpha)\right)^{n} \left(1 - \text{logit}^{-1}(\alpha)\right)^{N-n}. \end{aligned}$$

Log probability mass function

$$\begin{aligned}\log \mathrm{BinomialLogit}(n \mid N, \alpha) &= \log \Gamma(N+1) - \log \Gamma(n+1) - \log \Gamma(N-n+1) \\ &\quad + n \log \mathrm{logit}^{-1}(\alpha) + (N-n) \log \left(1 - \mathrm{logit}^{-1}(\alpha)\right),\end{aligned}$$

Gradient of log probability mass function

$$\frac{\partial}{\partial \alpha} \log \mathrm{BinomialLogit}(n \mid N, \alpha) = \frac{n}{\mathrm{logit}^{-1}(-\alpha)} - \frac{N-n}{\mathrm{logit}^{-1}(\alpha)}$$

Distribution statement

`n ~` **`binomial_logit`**`(N, alpha)`

Increment target log probability density with `binomial_logit_lupmf(n | N, alpha)`.

Available since 2.0

Stan functions

`real` **`binomial_logit_lpmf`**`(ints n | ints N, reals alpha)`
The log binomial probability mass of n successes in N trials given logit-scaled chance of success alpha

Available since 2.12

`real` **`binomial_logit_lupmf`**`(ints n | ints N, reals alpha)`
The log binomial probability mass of n successes in N trials given logit-scaled chance of success alpha dropping constant additive terms

Available since 2.25

16.3. Binomial-logit generalized linear model (Logistic Regression)

Stan also supplies a single function for a generalized linear model with binomial distribution and logit link function, i.e., a function for logistic regression with aggregated outcomes. This provides a more efficient implementation of logistic regression than a manually written regression in terms of a binomial distribution and matrix multiplication.

Probability mass function

Suppose $N \in \mathbb{N}$, $x \in \mathbb{R}^{n \cdot m}$, $\alpha \in \mathbb{R}^n$, $\beta \in \mathbb{R}^m$, and $n \in \{0, \ldots, N\}$. Then

$$\text{BinomialLogitGLM}(n \mid N, x, \alpha, \beta) = \text{Binomial}(n \mid N, \text{logit}^{-1}(\alpha_i + x_i \cdot \beta))$$
$$= \binom{N}{n} \left(\text{logit}^{-1}(\alpha_i + \sum_{1 \le j \le m} x_{ij} \cdot \beta_j) \right)^n \left(1 - \text{logit}^{-1}(\alpha_i + \sum_{1 \le j \le m} x_{ij} \cdot \beta_j) \right)^{N-n} .$$

Distribution statement

`n ~ binomial_logit_glm(N, x, alpha, beta)`

Increment target log probability density with `binomial_logit_glm_lupmf(n | N, x, alpha, beta)`.

Available since 2.34

Stan Functions

`real` **`binomial_logit_glm_lpmf`**`(int n | int N, matrix x, real alpha, vector beta)`
The log binomial probability mass of n given N trials and chance of success `inv_logit(alpha + x * beta)`.

Available since 2.34

`real` **`binomial_logit_glm_lupmf`**`(int n | int N, matrix x, real alpha, vector beta)`
The log binomial probability mass of n given N trials and chance of success `inv_logit(alpha + x * beta)` dropping constant additive terms.

Available since 2.34

`real` **`binomial_logit_glm_lpmf`**`(int n | int N, matrix x, vector alpha, vector beta)`
The log binomial probability mass of n given N trials and chance of success `inv_logit(alpha + x * beta)`.

Available since 2.34

`real` **`binomial_logit_glm_lupmf`**`(int n | int N, matrix x, vector alpha, vector beta)`
The log binomial probability mass of n given N trials and chance of success `inv_logit(alpha + x * beta)` dropping constant additive terms.

Available since 2.34

`real` **`binomial_logit_glm_lpmf`**`(array[] int n | array[] int N, row_vector x, real alpha, vector beta)`
The log binomial probability mass of n given N trials and chance of success `inv_logit(alpha + x * beta)`.

Available since 2.34

`real` **`binomial_logit_glm_lupmf`**`(array[] int n | array[] int N, row_vector x, real alpha, vector beta)`
The log binomial probability mass of n given N trials and chance of success `inv_logit(alpha + x * beta)` dropping constant additive terms.

Available since 2.34

`real` **`binomial_logit_glm_lpmf`**`(array[] int n | array[] int N, row_vector x, vector alpha, vector beta)`
The log binomial probability mass of n given N trials and chance of success `inv_logit(alpha + x * beta)`.

Available since 2.34

`real` **`binomial_logit_glm_lupmf`**`(array[] int n | array[] int N, row_vector x, vector alpha, vector beta)`
The log binomial probability mass of n given N trials and chance of success `inv_logit(alpha + x * beta)` dropping constant additive terms.

Available since 2.34

`real` **`binomial_logit_glm_lpmf`**`(array[] int n | array[] int N, matrix x, real alpha, vector beta)`
The log binomial probability mass of n given N trials and chance of success `inv_logit(alpha + x * beta)`.

Available since 2.34

`real` **`binomial_logit_glm_lupmf`**`(array[] int n | array[] int N, matrix x, real alpha, vector beta)`
The log binomial probability mass of n given N trials and chance of success `inv_logit(alpha + x * beta)` dropping constant additive terms.

Available since 2.34

`real` **`binomial_logit_glm_lpmf`**`(array[] int n | array[] int N, matrix x, vector alpha, vector beta)`

The log binomial probability mass of n given N trials and chance of success `inv_logit(alpha + x * beta)`.

Available since 2.34

`real` **`binomial_logit_glm_lupmf`**`(array[] int n | array[] int N, matrix x, vector alpha, vector beta)`
The log binomial probability mass of n given N trials and chance of success `inv_logit(alpha + x * beta)` dropping constant additive terms.

Available since 2.34

16.4. Beta-binomial distribution

Probability mass function

If $N \in \mathbb{N}$, $\alpha \in \mathbb{R}^+$, and $\beta \in \mathbb{R}^+$, then for $n \in 0, \ldots, N$,

$$\text{BetaBinomial}(n \mid N, \alpha, \beta) = \binom{N}{n} \frac{\mathrm{B}(n+\alpha, N-n+\beta)}{\mathrm{B}(\alpha, \beta)},$$

where the beta function $\mathrm{B}(u, v)$ is defined for $u \in \mathbb{R}^+$ and $v \in \mathbb{R}^+$ by

$$\mathrm{B}(u, v) = \frac{\Gamma(u)\,\Gamma(v)}{\Gamma(u+v)}.$$

Distribution statement

`n ~` **`beta_binomial`**`(N, alpha, beta)`

Increment target log probability density with `beta_binomial_lupmf(n | N, alpha, beta)`.

Available since 2.0

Stan functions

`real` **`beta_binomial_lpmf`**`(ints n | ints N, reals alpha, reals beta)`
The log beta-binomial probability mass of n successes in N trials given prior success count (plus one) of alpha and prior failure count (plus one) of beta

Available since 2.12

`real` **`beta_binomial_lupmf`**`(ints n | ints N, reals alpha, reals beta)`
The log beta-binomial probability mass of n successes in N trials given prior success count (plus one) of alpha and prior failure count (plus one) of beta dropping constant additive terms

Available since 2.25

`real` **`beta_binomial_cdf`**`(ints n | ints N, reals alpha, reals beta)`
The beta-binomial cumulative distribution function of n successes in N trials given prior success count (plus one) of alpha and prior failure count (plus one) of beta

Available since 2.0

`real` **`beta_binomial_lcdf`**`(ints n | ints N, reals alpha, reals beta)`
The log of the beta-binomial cumulative distribution function of n successes in N trials given prior success count (plus one) of alpha and prior failure count (plus one) of beta

Available since 2.12

`real` **`beta_binomial_lccdf`**`(ints n | ints N, reals alpha, reals beta)`
The log of the beta-binomial complementary cumulative distribution function of n successes in N trials given prior success count (plus one) of alpha and prior failure count (plus one) of beta

Available since 2.12

`R` **`beta_binomial_rng`**`(ints N, reals alpha, reals beta)`
Generate a beta-binomial variate with N trials, prior success count (plus one) of alpha, and prior failure count (plus one) of beta; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.18

16.5. Hypergeometric distribution

Probability mass function

If $a \in \mathbb{N}$, $b \in \mathbb{N}$, and $N \in \{0, \ldots, a+b\}$, then for $n \in \{\max(0, N - b), \ldots, \min(a, N)\}$,

$$\text{Hypergeometric}(n \mid N, a, b) = \frac{\binom{a}{n}\binom{b}{N-n}}{\binom{a+b}{N}}.$$

Distribution statement

`n ~` **`hypergeometric`**`(N, a, b)`

Increment target log probability density with `hypergeometric_lupmf(n | N, a, b)`.

Available since 2.0

Stan functions

`real` **`hypergeometric_lpmf`**`(int n | int N, int a, int b)`
The log hypergeometric probability mass of n successes in N trials given total success count of a and total failure count of b

Available since 2.12

`real` **`hypergeometric_lupmf`**`(int n | int N, int a, int b)`
The log hypergeometric probability mass of n successes in N trials given total success count of a and total failure count of b dropping constant additive terms

Available since 2.25

`int` **`hypergeometric_rng`**`(int N, int a, int b)`
Generate a hypergeometric variate with N trials, total success count of a, and total failure count of b; may only be used in transformed data and generated quantities blocks

Available since 2.18

16.6. Categorical distribution

Probability mass functions

If $N \in \mathbb{N}$, $N > 0$, and if $\theta \in \mathbb{R}^N$ forms an N-simplex (i.e., has nonnegative entries summing to one), then for $y \in \{1, \ldots, N\}$,

$$\text{Categorical}(y \mid \theta) = \theta_y.$$

In addition, Stan provides a log-odds scaled categorical distribution,

$$\text{CategoricalLogit}(y \mid \beta) = \text{Categorical}(y \mid \text{softmax}(\beta)).$$

See the definition of softmax for the definition of the softmax function.

Distribution statement

`y ~` **`categorical`**`(theta)`

Increment target log probability density with `categorical_lupmf(y | theta)` dropping constant additive terms.

Available since 2.0

Distribution statement

`y ~ `**`categorical_logit`**`(beta)`

Increment target log probability density with `categorical_logit_lupmf(y | beta)`.

Available since 2.4

Stan functions

All of the categorical distributions are vectorized so that the outcome y can be a single integer (type `int`) or an array of integers (type `array[] int`).

`real `**`categorical_lpmf`**`(ints y | vector theta)`
The log categorical probability mass function with outcome(s) y in $1 : N$ given N-vector of outcome probabilities theta. The parameter theta must have non-negative entries that sum to one, but it need not be a variable declared as a simplex.

Available since 2.12

`real `**`categorical_lupmf`**`(ints y | vector theta)`
The log categorical probability mass function with outcome(s) y in $1 : N$ given N-vector of outcome probabilities theta dropping constant additive terms. The parameter theta must have non-negative entries that sum to one, but it need not be a variable declared as a simplex.

Available since 2.25

`real `**`categorical_logit_lpmf`**`(ints y | vector beta)`
The log categorical probability mass function with outcome(s) y in $1 : N$ given log-odds of outcomes beta.

Available since 2.12

`real `**`categorical_logit_lupmf`**`(ints y | vector beta)`
The log categorical probability mass function with outcome(s) y in $1 : N$ given log-odds of outcomes beta dropping constant additive terms.

Available since 2.25

`int `**`categorical_rng`**`(vector theta)`
Generate a categorical variate with N-simplex distribution parameter theta; may only be used in transformed data and generated quantities blocks

Available since 2.0

`int` **`categorical_logit_rng`**`(vector beta)`
Generate a categorical variate with outcome in range $1 : N$ from log-odds vector beta; may only be used in transformed data and generated quantities blocks

Available since 2.16

16.7. Categorical logit generalized linear model (softmax regression)

Stan also supplies a single function for a generalized linear model with categorical distribution and logit link function, i.e. a function for a softmax regression. This provides a more efficient implementation of softmax regression than a manually written regression in terms of a categorical distribution and matrix multiplication.

Note that the implementation does not put any restrictions on the coefficient matrix β. It is up to the user to use a reference category, a suitable prior or some other means of identifiability. See Multi-logit in the Stan User's Guide.

Probability mass functions

If $N, M, K \in \mathbb{N}$, $N, M, K > 0$, and if $x \in \mathbb{R}^{M \times K}, \alpha \in \mathbb{R}^N, \beta \in \mathbb{R}^{K \cdot N}$, then for $y \in \{1, \ldots, N\}^M$,

$$\begin{aligned} \text{CategoricalLogitGLM}(y \mid x, \alpha, \beta) &= \prod_{1 \le i \le M} \text{CategoricalLogit}(y_i \mid \alpha + x_i \cdot \beta) \\ &= \prod_{1 \le i \le M} \text{Categorical}(y_i \mid softmax(\alpha + x_i \cdot \beta)). \end{aligned}$$

See the definition of softmax for the definition of the softmax function.

Distribution statement

`y ~` **`categorical_logit_glm`**`(x, alpha, beta)`

Increment target log probability density with `categorical_logit_glm_lupmf(y | x, alpha, beta)`.

Available since 2.23

Stan functions

`real` **`categorical_logit_glm_lpmf`**`(int y | row_vector x, vector alpha, matrix beta)`
The log categorical probability mass function with outcome `y` in $1 : N$ given N-vector of log-odds of outcomes `alpha + x * beta`.

Available since 2.23

`real` **`categorical_logit_glm_lupmf`**`(int y | row_vector x, vector alpha, matrix beta)`
The log categorical probability mass function with outcome `y` in $1 : N$ given N-vector of log-odds of outcomes `alpha + x * beta` dropping constant additive terms.

Available since 2.25

`real` **`categorical_logit_glm_lpmf`**`(int y | matrix x, vector alpha, matrix beta)`
The log categorical probability mass function with outcomes `y` in $1 : N$ given N-vector of log-odds of outcomes `alpha + x * beta`.

Available since 2.23

`real` **`categorical_logit_glm_lupmf`**`(int y | matrix x, vector alpha, matrix beta)`
The log categorical probability mass function with outcomes `y` in $1 : N$ given N-vector of log-odds of outcomes `alpha + x * beta` dropping constant additive terms.

Available since 2.25

`real` **`categorical_logit_glm_lpmf`**`(array[] int y | row_vector x, vector alpha, matrix beta)`
The log categorical probability mass function with outcomes `y` in $1 : N$ given N-vector of log-odds of outcomes `alpha + x * beta`.

Available since 2.23

`real` **`categorical_logit_glm_lupmf`**`(array[] int y | row_vector x, vector alpha, matrix beta)`
The log categorical probability mass function with outcomes `y` in $1 : N$ given N-vector of log-odds of outcomes `alpha + x * beta` dropping constant additive terms.

Available since 2.25

`real` **`categorical_logit_glm_lpmf`**`(array[] int y | matrix x, vector alpha, matrix beta)`
The log categorical probability mass function with outcomes `y` in $1 : N$ given N-vector of log-odds of outcomes `alpha + x * beta`.

Available since 2.23

`real` **`categorical_logit_glm_lupmf`**`(array[] int y | matrix x, vector alpha, matrix beta)`
The log categorical probability mass function with outcomes `y` in $1 : N$ given N-vector of log-odds of outcomes `alpha + x * beta` dropping constant additive terms.

Available since 2.25

16.8. Discrete range distribution

Probability mass functions

If $l, u \in \mathbb{Z}$ are lower and upper bounds ($l \leq u$), then for any integer $y \in \{l, \ldots, u\}$,

$$\text{DiscreteRange}(y \mid l, u) = \frac{1}{u - l + 1}.$$

Distribution statement

`y ~` **`discrete_range`**`(l, u)`

Increment the target log probability density with `discrete_range_lupmf(y | l, u)` dropping constant additive terms.

Available since 2.26

Stan functions

All of the discrete range distributions are vectorized so that the outcome `y` and the bounds `l, u` can be a single integer (type `int`) or an array of integers (type `array[] int`).

`real` **`discrete_range_lpmf`**`(ints y | ints l, ints u)`
The log probability mass function with outcome(s) y in $l : u$.

Available since 2.26

`real` **`discrete_range_lupmf`**`(ints y | ints l, ints u)`
The log probability mass function with outcome(s) y in $l : u$ dropping constant additive terms.

Available since 2.26

`real` **`discrete_range_cdf`**`(ints y | ints l, ints u)`
The discrete range cumulative distribution function for the given y, lower and upper bounds.

Available since 2.26

`real` **`discrete_range_lcdf`**`(ints y | ints l, ints u)`
The log of the discrete range cumulative distribution function for the given y, lower and upper bounds.

Available since 2.26

`real` **`discrete_range_lccdf`**`(ints y | ints l, ints u)`
The log of the discrete range complementary cumulative distribution function for the given y, lower and upper bounds.

Available since 2.26

`ints` **`discrete_range_rng`**`(ints l, ints u)`
Generate a discrete variate between the given lower and upper bounds; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.26

16.9. Ordered logistic distribution

Probability mass function

If $K \in \mathbb{N}$ with $K > 2$, $c \in \mathbb{R}^{K-1}$ such that $c_k < c_{k+1}$ for $k \in \{1, \ldots, K-2\}$, and $\eta \in \mathbb{R}$, then for $k \in \{1, \ldots, K\}$,

$$\text{OrderedLogistic}(k \mid \eta, c) = \begin{cases} 1 - \text{logit}^{-1}(\eta - c_1) & \text{if } k = 1, \\ \text{logit}^{-1}(\eta - c_{k-1}) - \text{logit}^{-1}(\eta - c_k) & \text{if } 1 < k < K, \text{and} \\ \text{logit}^{-1}(\eta - c_{K-1}) - 0 & \text{if } k = K. \end{cases}$$

The $k = K$ case is written with the redundant subtraction of zero to illustrate the parallelism of the cases; the $k = 1$ and $k = K$ edge cases can be subsumed into the general definition by setting $c_0 = -\infty$ and $c_K = +\infty$ with $\text{logit}^{-1}(-\infty) = 0$ and $\text{logit}^{-1}(\infty) = 1$.

Distribution statement

`k ~` **`ordered_logistic`**`(eta, c)`

Increment target log probability density with `ordered_logistic_lupmf(k | eta, c)`.

Available since 2.0

Stan functions

`real` **`ordered_logistic_lpmf`**`(ints k | vector eta, vectors c)`
The log ordered logistic probability mass of k given linear predictors `eta`, and cutpoints `c`.

Available since 2.18

`real` **`ordered_logistic_lupmf`**`(ints k | vector eta, vectors c)`
The log ordered logistic probability mass of k given linear predictors `eta`, and cutpoints `c` dropping constant additive terms.

Available since 2.25

`int` **`ordered_logistic_rng`**`(real eta, vector c)`
Generate an ordered logistic variate with linear predictor `eta` and cutpoints `c`; may only be used in transformed data and generated quantities blocks

Available since 2.0

16.10. Ordered logistic generalized linear model (ordinal regression)

Probability mass function

If $N, M, K \in \mathbb{N}$ with $N, M > 0$, $K > 2$, $c \in \mathbb{R}^{K-1}$ such that $c_k < c_{k+1}$ for $k \in \{1, \ldots, K-2\}$, and $x \in \mathbb{R}^{N \times M}$, $\beta \in \mathbb{R}^M$, then for $y \in \{1, \ldots, K\}^N$,

$$
\begin{aligned}
&\text{OrderedLogisticGLM}(y \mid x, \beta, c) \\
&= \prod_{1 \le i \le N} \text{OrderedLogistic}(y_i \mid x_i \cdot \beta, c) \\
&= \prod_{1 \le i \le N} \begin{cases} 1 - \text{logit}^{-1}(x_i \cdot \beta - c_1) & \text{if } y = 1, \\ \text{logit}^{-1}(x_i \cdot \beta - c_{y-1}) - \text{logit}^{-1}(x_i \cdot \beta - c_y) & \text{if } 1 < y < K, \text{and} \\ \text{logit}^{-1}(x_i \cdot \beta - c_{K-1}) - 0 & \text{if } y = K. \end{cases}
\end{aligned}
$$

The $k = K$ case is written with the redundant subtraction of zero to illustrate the parallelism of the cases; the $y = 1$ and $y = K$ edge cases can be subsumed into the general definition by setting $c_0 = -\infty$ and $c_K = +\infty$ with $\text{logit}^{-1}(-\infty) = 0$ and $\text{logit}^{-1}(\infty) = 1$.

Distribution statement

`y ~` **`ordered_logistic_glm`**`(x, beta, c)`

Increment target log probability density with `ordered_logistic_lupmf(y | x, beta, c)`.

Available since 2.23

Stan functions

`real` **`ordered_logistic_glm_lpmf`**`(int y | row_vector x, vector beta, vector c)`
The log ordered logistic probability mass of y, given linear predictors `x * beta`, and cutpoints c. The cutpoints c must be ordered.

Available since 2.23

`real` **`ordered_logistic_glm_lupmf`**`(int y | row_vector x, vector beta, vector c)`
The log ordered logistic probability mass of y, given linear predictors `x * beta`, and cutpoints c dropping constant additive terms. The cutpoints c must be ordered.

Available since 2.25

`real` **`ordered_logistic_glm_lpmf`**`(int y | matrix x, vector beta, vector c)`
The log ordered logistic probability mass of y, given linear predictors `x * beta`, and cutpoints c. The cutpoints c must be ordered.

Available since 2.23

`real` **`ordered_logistic_glm_lupmf`**`(int y | matrix x, vector beta, vector c)`
The log ordered logistic probability mass of y, given linear predictors `x * beta`, and cutpoints c dropping constant additive terms. The cutpoints c must be ordered.

Available since 2.25

`real` **`ordered_logistic_glm_lpmf`**`(array[] int y | row_vector x, vector beta, vector c)`
The log ordered logistic probability mass of y, given linear predictors `x * beta`, and cutpoints c. The cutpoints c must be ordered.

Available since 2.23

`real` **`ordered_logistic_glm_lupmf`**`(array[] int y | row_vector x, vector beta, vector c)`
The log ordered logistic probability mass of y, given linear predictors `x * beta`, and cutpoints c dropping constant additive terms. The cutpoints c must be ordered.

Available since 2.25

`real` **`ordered_logistic_glm_lpmf`**`(array[] int y | matrix x, vector beta, vector c)`
The log ordered logistic probability mass of y, given linear predictors `x * beta`, and cutpoints c. The cutpoints `c` must be ordered.

Available since 2.23

`real` **`ordered_logistic_glm_lupmf`**`(array[] int y | matrix x, vector beta, vector c)`
The log ordered logistic probability mass of y, given linear predictors `x * beta`, and cutpoints c dropping constant additive terms. The cutpoints `c` must be ordered.

Available since 2.25

16.11. Ordered probit distribution

Probability mass function

If $K \in \mathbb{N}$ with $K > 2$, $c \in \mathbb{R}^{K-1}$ such that $c_k < c_{k+1}$ for $k \in \{1, \ldots, K-2\}$, and $\eta \in \mathbb{R}$, then for $k \in \{1, \ldots, K\}$,

$$\text{OrderedProbit}(k \mid \eta, c) = \begin{cases} 1 - \Phi(\eta - c_1) & \text{if } k = 1, \\ \Phi(\eta - c_{k-1}) - \Phi(\eta - c_k) & \text{if } 1 < k < K, \text{and} \\ \Phi(\eta - c_{K-1}) - 0 & \text{if } k = K. \end{cases}$$

The $k = K$ case is written with the redundant subtraction of zero to illustrate the parallelism of the cases; the $k = 1$ and $k = K$ edge cases can be subsumed into the general definition by setting $c_0 = -\infty$ and $c_K = +\infty$ with $\Phi(-\infty) = 0$ and $\Phi(\infty) = 1$.

Distribution statement

`k ~` **`ordered_probit`**`(eta, c)`

Increment target log probability density with `ordered_probit_lupmf(k | eta, c)`.

Available since 2.19

Stan functions

`real` **`ordered_probit_lpmf`**`(ints k | vector eta, vectors c)`
The log ordered probit probability mass of k given linear predictors eta, and cutpoints c.

Available since 2.18

`real` **`ordered_probit_lupmf`**`(ints k | vector eta, vectors c)`
The log ordered probit probability mass of k given linear predictors eta, and cutpoints c dropping constant additive terms.

Available since 2.25

`real` **`ordered_probit_lpmf`**`(ints k | real eta, vectors c)`
The log ordered probit probability mass of k given linear predictor eta, and cutpoints c.

Available since 2.19

`real` **`ordered_probit_lupmf`**`(ints k | real eta, vectors c)`
The log ordered probit probability mass of k given linear predictor eta, and cutpoints c dropping constant additive terms.

Available since 2.19

`int` **`ordered_probit_rng`**`(real eta, vector c)`
Generate an ordered probit variate with linear predictor eta and cutpoints c; may only be used in transformed data and generated quantities blocks

Available since 2.18

17. Unbounded Discrete Distributions

The unbounded discrete distributions have support over the natural numbers (i.e., the non-negative integers).

17.1. Negative binomial distribution

For the negative binomial distribution Stan uses the parameterization described in Gelman et al. (2013). For alternative parameterizations, see section negative binomial glm.

Probability mass function

If $\alpha \in \mathbb{R}^+$ and $\beta \in \mathbb{R}^+$, then for $n \in \mathbb{N}$,

$$\text{NegBinomial}(n \mid \alpha, \beta) = \binom{n + \alpha - 1}{\alpha - 1} \left(\frac{\beta}{\beta + 1}\right)^{\alpha} \left(\frac{1}{\beta + 1}\right)^{n}.$$

The mean and variance of a random variable $n \sim \text{NegBinomial}(\alpha, \beta)$ are given by

$$\mathbb{E}[n] = \frac{\alpha}{\beta} \quad \text{and} \quad \text{Var}[n] = \frac{\alpha}{\beta^2}(\beta + 1).$$

Distribution statement

`n ~ `**`neg_binomial`**`(alpha, beta)`

Increment target log probability density with `neg_binomial_lupmf(n | alpha, beta)`.

Available since 2.0

Stan functions

`real` **`neg_binomial_lpmf`**`(ints n | reals alpha, reals beta)`
The log negative binomial probability mass of `n` given shape `alpha` and inverse scale `beta`

Available since 2.12

`real` **`neg_binomial_lupmf`**`(ints n | reals alpha, reals beta)`
The log negative binomial probability mass of `n` given shape `alpha` and inverse scale `beta` dropping constant additive terms

Available since 2.25

`real` **`neg_binomial_cdf`**`(ints n | reals alpha, reals beta)`
The negative binomial cumulative distribution function of `n` given shape `alpha` and inverse scale `beta`

Available since 2.0

`real` **`neg_binomial_lcdf`**`(ints n | reals alpha, reals beta)`
The log of the negative binomial cumulative distribution function of `n` given shape `alpha` and inverse scale `beta`

Available since 2.12

`real` **`neg_binomial_lccdf`**`(ints n | reals alpha, reals beta)`
The log of the negative binomial complementary cumulative distribution function of `n` given shape `alpha` and inverse scale `beta`

Available since 2.12

`R` **`neg_binomial_rng`**`(reals alpha, reals beta)`
Generate a negative binomial variate with shape `alpha` and inverse scale `beta`; may only be used in transformed data and generated quantities blocks. `alpha / beta` must be less than 2^{29}. For a description of argument and return types, see section vectorized function signatures.

Available since 2.18

17.2. Negative binomial distribution (alternative parameterization)

Stan also provides an alternative parameterization of the negative binomial distribution directly using a mean (i.e., location) parameter and a parameter that controls overdispersion relative to the square of the mean. Section combinatorial functions, below, provides a second alternative parameterization directly in terms of the log mean.

Probability mass function

The first parameterization is for $\mu \in \mathbb{R}^+$ and $\phi \in \mathbb{R}^+$, which for $n \in \mathbb{N}$ is defined as

$$\text{NegBinomial2}(n \mid \mu, \phi) = \binom{n+\phi-1}{n} \left(\frac{\mu}{\mu+\phi}\right)^n \left(\frac{\phi}{\mu+\phi}\right)^\phi.$$

The mean and variance of a random variable $n \sim \text{NegBinomial2}(n \mid \mu, \phi)$ are

$$\mathbb{E}[n] = \mu \quad \text{and} \quad \text{Var}[n] = \mu + \frac{\mu^2}{\phi}.$$

Recall that $\text{Poisson}(\mu)$ has variance μ, so $\mu^2/\phi > 0$ is the additional variance of the negative binomial above that of the Poisson with mean μ. So the inverse of parameter ϕ controls the overdispersion, scaled by the square of the mean, μ^2.

Distribution statement

`n ~ neg_binomial_2(mu, phi)`

Increment target log probability density with `neg_binomial_2_lupmf(n | mu, phi)`.

Available since 2.3

Stan functions

`real neg_binomial_2_lpmf(ints n | reals mu, reals phi)`
The log negative binomial probability mass of `n` given location `mu` and precision `phi`.

Available since 2.20

`real neg_binomial_2_lupmf(ints n | reals mu, reals phi)`
The log negative binomial probability mass of `n` given location `mu` and precision `phi` dropping constant additive terms.

Available since 2.25

`real neg_binomial_2_cdf(ints n | reals mu, reals phi)`
The negative binomial cumulative distribution function of `n` given location `mu` and precision `phi`.

Available since 2.6

`real neg_binomial_2_lcdf(ints n | reals mu, reals phi)`
The log of the negative binomial cumulative distribution function of `n` given location `mu` and precision `phi`.

Available since 2.12

`real neg_binomial_2_lccdf(ints n | reals mu, reals phi)`
The log of the negative binomial complementary cumulative distribution function of `n` given location `mu` and precision `phi`.

Available since 2.12

`R` **`neg_binomial_2_rng`**`(reals mu, reals phi)`
Generate a negative binomial variate with location `mu` and precision `phi`; may only be used in transformed data and generated quantities blocks. `mu` must be less than 2^{29}. For a description of argument and return types, see section vectorized function signatures.

Available since 2.18

17.3. Negative binomial distribution (log alternative parameterization)

Related to the parameterization in section negative binomial, alternative parameterization, the following parameterization uses a log mean parameter $\eta = \log(\mu)$, defined for $\eta \in \mathbb{R}$, $\phi \in \mathbb{R}^+$, so that for $n \in \mathbb{N}$,

$$\text{NegBinomial2Log}(n \mid \eta, \phi) = \text{NegBinomial2}(n | \exp(\eta), \phi).$$

This alternative may be used for sampling, as a function, and for random number generation, but as of yet, there are no CDFs implemented for it. This is especially useful for log-linear negative binomial regressions.

Distribution statement

`n ~` **`neg_binomial_2_log`**`(eta, phi)`

Increment target log probability density with `neg_binomial_2_log_lupmf(n | eta, phi)`.

Available since 2.3

Stan functions

`real` **`neg_binomial_2_log_lpmf`**`(ints n | reals eta, reals phi)`
The log negative binomial probability mass of `n` given log-location `eta` and inverse overdispersion parameter `phi`.

Available since 2.20

`real` **`neg_binomial_2_log_lupmf`**`(ints n | reals eta, reals phi)`
The log negative binomial probability mass of `n` given log-location `eta` and inverse overdispersion parameter `phi` dropping constant additive terms.

Available since 2.25

`R` **`neg_binomial_2_log_rng`**`(reals eta, reals phi)`
Generate a negative binomial variate with log-location `eta` and inverse overdisper-

sion control phi; may only be used in transformed data and generated quantities blocks. eta must be less than $29 \log 2$. For a description of argument and return types, see section vectorized function signatures.

Available since 2.18

17.4. Negative-binomial-2-log generalized linear model (negative binomial regression)

Stan also supplies a single function for a generalized linear model with negative binomial distribution and log link function, i.e. a function for a negative binomial regression. This provides a more efficient implementation of negative binomial regression than a manually written regression in terms of a negative binomial distribution and matrix multiplication.

Probability mass function

If $x \in \mathbb{R}^{n \cdot m}, \alpha \in \mathbb{R}^n, \beta \in \mathbb{R}^m, \phi \in \mathbb{R}^+$, then for $y \in \mathbb{N}^n$,

$$\text{NegBinomial2LogGLM}(y \mid x, \alpha, \beta, \phi) = \prod_{1 \le i \le n} \text{NegBinomial2}(y_i \mid \exp(\alpha_i + x_i \cdot \beta), \phi).$$

Distribution statement

y ~ **neg_binomial_2_log_glm**(x, alpha, beta, phi)

Increment target log probability density with neg_binomial_2_log_glm_lupmf(y | x, alpha, beta, phi).

Available since 2.19

Stan functions

real **neg_binomial_2_log_glm_lpmf**(int y | matrix x, real alpha, vector beta, real phi)
The log negative binomial probability mass of y given log-location alpha + x * beta and inverse overdispersion parameter phi.

Available since 2.23

real **neg_binomial_2_log_glm_lupmf**(int y | matrix x, real alpha, vector beta, real phi)
The log negative binomial probability mass of y given log-location alpha + x * beta and inverse overdispersion parameter phi dropping constant additive terms.

Available since 2.25

`real` **`neg_binomial_2_log_glm_lpmf`**`(int y | matrix x, vector alpha, vector beta, real phi)`
The log negative binomial probability mass of `y` given log-location `alpha + x * beta` and inverse overdispersion parameter `phi`.

Available since 2.23

`real` **`neg_binomial_2_log_glm_lupmf`**`(int y | matrix x, vector alpha, vector beta, real phi)`
The log negative binomial probability mass of `y` given log-location `alpha + x * beta` and inverse overdispersion parameter `phi` dropping constant additive terms.

Available since 2.25

`real` **`neg_binomial_2_log_glm_lpmf`**`(array[] int y | row_vector x, real alpha, vector beta, real phi)`
The log negative binomial probability mass of `y` given log-location `alpha + x * beta` and inverse overdispersion parameter `phi`.

Available since 2.23

`real` **`neg_binomial_2_log_glm_lupmf`**`(array[] int y | row_vector x, real alpha, vector beta, real phi)`
The log negative binomial probability mass of `y` given log-location `alpha + x * beta` and inverse overdispersion parameter `phi` dropping constant additive terms.

Available since 2.25

`real` **`neg_binomial_2_log_glm_lpmf`**`(array[] int y | row_vector x, vector alpha, vector beta, real phi)`
The log negative binomial probability mass of `y` given log-location `alpha + x * beta` and inverse overdispersion parameter `phi`.

Available since 2.23

`real` **`neg_binomial_2_log_glm_lupmf`**`(array[] int y | row_vector x, vector alpha, vector beta, real phi)`
The log negative binomial probability mass of `y` given log-location `alpha + x * beta` and inverse overdispersion parameter `phi` dropping constant additive terms.

Available since 2.25

`real` **`neg_binomial_2_log_glm_lpmf`**`(array[] int y | matrix x, real alpha, vector beta, real phi)`

The log negative binomial probability mass of `y` given log-location `alpha + x * beta` and inverse overdispersion parameter `phi`.

Available since 2.18

`real` **`neg_binomial_2_log_glm_lupmf`**`(array[] int y | matrix x, real alpha, vector beta, real phi)`
The log negative binomial probability mass of `y` given log-location `alpha + x * beta` and inverse overdispersion parameter `phi` dropping constant additive terms.

Available since 2.25

`real` **`neg_binomial_2_log_glm_lpmf`**`(array[] int y | matrix x, vector alpha, vector beta, real phi)`
The log negative binomial probability mass of `y` given log-location `alpha + x * beta` and inverse overdispersion parameter `phi`.

Available since 2.18

`real` **`neg_binomial_2_log_glm_lupmf`**`(array[] int y | matrix x, vector alpha, vector beta, real phi)`
The log negative binomial probability mass of `y` given log-location `alpha + x * beta` and inverse overdispersion parameter `phi` dropping constant additive terms.

Available since 2.25

17.5. Poisson distribution

Probability mass function

If $\lambda \in \mathbb{R}^+$, then for $n \in \mathbb{N}$,

$$\text{Poisson}(n|\lambda) = \frac{1}{n!}\,\lambda^n\,\exp(-\lambda).$$

Distribution statement

`n ~` **`poisson`**`(lambda)`

Increment target log probability density with `poisson_lupmf(n | lambda)`.

Available since 2.0

Stan functions

`real` **`poisson_lpmf`**`(ints n | reals lambda)`
The log Poisson probability mass of n given rate lambda

Available since 2.12

`real` **`poisson_lupmf`**`(ints n | reals lambda)`
The log Poisson probability mass of n given rate lambda dropping constant additive terms

Available since 2.25

`real` **`poisson_cdf`**`(ints n | reals lambda)`
The Poisson cumulative distribution function of n given rate lambda

Available since 2.0

`real` **`poisson_lcdf`**`(ints n | reals lambda)`
The log of the Poisson cumulative distribution function of n given rate lambda

Available since 2.12

`real` **`poisson_lccdf`**`(ints n | reals lambda)`
The log of the Poisson complementary cumulative distribution function of n given rate lambda

Available since 2.12

`R` **`poisson_rng`**`(reals lambda)`
Generate a Poisson variate with rate lambda; may only be used in transformed data and generated quantities blocks. lambda must be less than 2^{30}. For a description of argument and return types, see section vectorized function signatures.

Available since 2.18

17.6. Poisson distribution, log parameterization

Stan also provides a parameterization of the Poisson using the log rate $\alpha = \log \lambda$ as a parameter. This is useful for log-linear Poisson regressions so that the predictor does not need to be exponentiated and passed into the standard Poisson probability function.

Probability mass function

If $\alpha \in \mathbb{R}$, then for $n \in \mathbb{N}$,

$$\text{PoissonLog}(n|\alpha) = \frac{1}{n!} \exp\left(n\alpha - \exp(\alpha)\right).$$

Distribution statement

`n ~` **`poisson_log`**`(alpha)`

Increment target log probability density with `poisson_log_lupmf(n | alpha)`.

Available since 2.0

Stan functions

`real` **`poisson_log_lpmf`**`(ints n | reals alpha)`
The log Poisson probability mass of n given log rate alpha

Available since 2.12

`real` **`poisson_log_lupmf`**`(ints n | reals alpha)`
The log Poisson probability mass of n given log rate alpha dropping constant additive terms

Available since 2.25

`R` **`poisson_log_rng`**`(reals alpha)`
Generate a Poisson variate with log rate alpha; may only be used in transformed data and generated quantities blocks. alpha must be less than $30 \log 2$. For a description of argument and return types, see section vectorized function signatures.

Available since 2.18

17.7. Poisson-log generalized linear model (Poisson regression)

Stan also supplies a single function for a generalized linear model with Poisson distribution and log link function, i.e. a function for a Poisson regression. This provides a more efficient implementation of Poisson regression than a manually written regression in terms of a Poisson distribution and matrix multiplication.

Probability mass function

If $x \in \mathbb{R}^{n \cdot m}, \alpha \in \mathbb{R}^n, \beta \in \mathbb{R}^m$, then for $y \in \mathbb{N}^n$,

$$\text{PoissonLogGLM}(y|x,\alpha,\beta) = \prod_{1 \le i \le n} \text{Poisson}(y_i | \exp(\alpha_i + x_i \cdot \beta)).$$

Distribution statement

`y ~` **`poisson_log_glm`**`(x, alpha, beta)`

Increment target log probability density with `poisson_log_glm_lupmf(y | x, alpha, beta)`.

Available since 2.19

Stan functions

`real` **`poisson_log_glm_lpmf`**`(int y | matrix x, real alpha, vector beta)`

The log Poisson probability mass of `y` given the log-rate `alpha + x * beta`.

Available since 2.23

`real` **`poisson_log_glm_lupmf`**`(int y | matrix x, real alpha, vector beta)`
The log Poisson probability mass of `y` given the log-rate `alpha + x * beta` dropping constant additive terms.

Available since 2.25

`real` **`poisson_log_glm_lpmf`**`(int y | matrix x, vector alpha, vector beta)`
The log Poisson probability mass of `y` given the log-rate `alpha + x * beta`.

Available since 2.23

`real` **`poisson_log_glm_lupmf`**`(int y | matrix x, vector alpha, vector beta)`
The log Poisson probability mass of `y` given the log-rate `alpha + x * beta` dropping constant additive terms.

Available since 2.25

`real` **`poisson_log_glm_lpmf`**`(array[] int y | row_vector x, real alpha, vector beta)`
The log Poisson probability mass of `y` given the log-rate `alpha + x * beta`.

Available since 2.23

`real` **`poisson_log_glm_lupmf`**`(array[] int y | row_vector x, real alpha, vector beta)`
The log Poisson probability mass of `y` given the log-rate `alpha + x * beta` dropping constant additive terms.

Available since 2.25

`real` **`poisson_log_glm_lpmf`**`(array[] int y | row_vector x, vector alpha, vector beta)`
The log Poisson probability mass of `y` given the log-rate `alpha + x * beta`.

Available since 2.23

`real` **`poisson_log_glm_lupmf`**`(array[] int y | row_vector x, vector alpha, vector beta)`

The log Poisson probability mass of `y` given the log-rate `alpha + x * beta` dropping constant additive terms.

Available since 2.25

`real` **`poisson_log_glm_lpmf`**`(array[] int y | matrix x, real alpha, vector beta)`
The log Poisson probability mass of `y` given the log-rate `alpha + x * beta`.

Available since 2.18

`real` **`poisson_log_glm_lupmf`**`(array[] int y | matrix x, real alpha, vector beta)`
The log Poisson probability mass of `y` given the log-rate `alpha + x * beta` dropping constant additive terms.

Available since 2.25

`real` **`poisson_log_glm_lpmf`**`(array[] int y | matrix x, vector alpha, vector beta)`
The log Poisson probability mass of `y` given the log-rate `alpha + x * beta`.

Available since 2.18

`real` **`poisson_log_glm_lupmf`**`(array[] int y | matrix x, vector alpha, vector beta)`
The log Poisson probability mass of `y` given the log-rate `alpha + x * beta` dropping constant additive terms.

Available since 2.25

18. Multivariate Discrete Distributions

The multivariate discrete distributions are over multiple integer values, which are expressed in Stan as arrays.

18.1. Multinomial distribution

Probability mass function

If $K \in \mathbb{N}$, $N \in \mathbb{N}$, and $\theta \in K$-simplex, then for $y \in \mathbb{N}^K$ such that $\sum_{k=1}^K y_k = N$,

$$\text{Multinomial}(y|\theta) = \binom{N}{y_1, \ldots, y_K} \prod_{k=1}^K \theta_k^{y_k},$$

where the multinomial coefficient is defined by

$$\binom{N}{y_1, \ldots, y_k} = \frac{N!}{\prod_{k=1}^K y_k!}.$$

Distribution statement

`y ~ multinomial(theta)`

Increment target log probability density with `multinomial_lupmf(y | theta)`.

Available since 2.0

Stan functions

`real multinomial_lpmf(array[] int y | vector theta)`
The log multinomial probability mass function with outcome array `y` of size K given the K-simplex distribution parameter theta and (implicit) total count `N = sum(y)`

Available since 2.12

`real multinomial_lupmf(array[] int y | vector theta)`
The log multinomial probability mass function with outcome array `y` of size K given the K-simplex distribution parameter theta and (implicit) total count `N = sum(y)` dropping constant additive terms

Available since 2.25

`array[] int` **`multinomial_rng`**`(vector theta, int N)`
Generate a multinomial variate with simplex distribution parameter theta and total count N; may only be used in transformed data and generated quantities blocks

Available since 2.8

18.2. Multinomial distribution, logit parameterization

Stan also provides a version of the multinomial probability mass function distribution with the K-simplex for the event count probabilities per category given on the unconstrained logistic scale.

Probability mass function

If $K \in \mathbb{N}$, $N \in \mathbb{N}$, and $\text{softmax}(\theta) \in K$-simplex, then for $y \in \mathbb{N}^K$ such that $\sum_{k=1}^K y_k = N$,

$$\begin{aligned}\text{MultinomialLogit}(y \mid \gamma) &= \text{Multinomial}(y \mid \text{softmax}(\gamma)) \\ &= \binom{N}{y_1, \ldots, y_K} \prod_{k=1}^K [\text{softmax}(\gamma_k)]^{y_k},\end{aligned}$$

where the multinomial coefficient is defined by

$$\binom{N}{y_1, \ldots, y_k} = \frac{N!}{\prod_{k=1}^K y_k!}.$$

Distribution statement

`y ~` **`multinomial_logit`**`(gamma)`

Increment target log probability density with `multinomial_logit_lupmf(y | gamma)`.

Available since 2.24

Stan functions

`real` **`multinomial_logit_lpmf`**`(array[] int y | vector gamma)`
The log multinomial probability mass function with outcome array `y` of size K given the log K-simplex distribution parameter γ and (implicit) total count `N = sum(y)`

Available since 2.24

`real` **`multinomial_logit_lupmf`**`(array[] int y | vector gamma)`
The log multinomial probability mass function with outcome array `y` of size K given

the log K-simplex distribution parameter γ and (implicit) total count `N = sum(y)` dropping constant additive terms

Available since 2.25

`array[] int` **`multinomial_logit_rng`**`(vector gamma, int N)`
Generate a variate from a multinomial distribution with probabilities `softmax(gamma)` and total count `N`; may only be used in transformed data and generated quantities blocks.

Available since 2.24

18.3. Dirichlet-multinomial distribution

Stan also provides the Dirichlet-multinomial distribution, which generalizes the Beta-binomial distribution to more than two categories. As such, it is an overdispersed version of the multinomial distribution.

Probability mass function

If $K \in \mathbb{N}$, $N \in \mathbb{N}$, and $\alpha \in \mathbb{R}_+^K$, then for $y \in \mathbb{N}^K$ such that $\sum_{k=1}^K y_k = N$, the PMF of the Dirichlet-multinomial distribution is defined as

$$\text{DirMult}(y|\theta) = \frac{\Gamma(\alpha_0)\Gamma(N+1)}{\Gamma(N+\alpha_0)} \prod_{k=1}^{K} \frac{\Gamma(y_k+\alpha_k)}{\Gamma(\alpha_k)\Gamma(y_k+1)},$$

where α_0 is defined as $\alpha_0 = \sum_{k=1}^K \alpha_k$.

Distribution statement

`y ~` **`dirichlet_multinomial`**`(alpha)`

Increment target log probability density with `dirichlet_multinomial_lupmf(y | alpha)`.

Available since 2.34

Stan functions

`real` **`dirichlet_multinomial_lpmf`**`(array[] int y | vector alpha)`
The log multinomial probability mass function with outcome array `y` with K elements given the positive K-vector distribution parameter `alpha` and (implicit) total count `N = sum(y)`.

Available since 2.34

`real` **`dirichlet_multinomial_lupmf`**`(array[] int y | vector alpha)`
The log multinomial probability mass function with outcome array `y` with K ele-

ments, given the positive K-vector distribution parameter `alpha` and (implicit) total count `N = sum(y)` dropping constant additive terms.

Available since 2.34

`array[] int` **`dirichlet_multinomial_rng`**`(vector alpha, int N)`
Generate a multinomial variate with positive vector distribution parameter `alpha` and total count `N`; may only be used in transformed data and generated quantities blocks. This is equivalent to `multinomial_rng(dirichlet_rng(alpha), N)`.

Available since 2.34

Part III

Continuous Distributions

19. Unbounded Continuous Distributions

The unbounded univariate continuous probability distributions have support on all real numbers.

19.1. Normal distribution

Probability density function

If $\mu \in \mathbb{R}$ and $\sigma \in \mathbb{R}^+$, then for $y \in \mathbb{R}$,

$$\text{Normal}(y|\mu, \sigma) = \frac{1}{\sqrt{2\pi}\,\sigma} \exp\left(-\frac{1}{2}\left(\frac{y-\mu}{\sigma}\right)^2\right).$$

Distribution statement

`y ~ normal(mu, sigma)`

Increment target log probability density with `normal_lupdf(y | mu, sigma)`.

Available since 2.0

Stan functions

`real normal_lpdf(reals y | reals mu, reals sigma)`
The log of the normal density of y given location mu and scale sigma

Available since 2.12

`real normal_lupdf(reals y | reals mu, reals sigma)`
The log of the normal density of y given location mu and scale sigma dropping constant additive terms.

Available since 2.25

`real normal_cdf(reals y | reals mu, reals sigma)`
The cumulative normal distribution of y given location mu and scale sigma; normal_cdf will underflow to 0 for $\frac{y-\mu}{\sigma}$ below -37.5 and overflow to 1 for $\frac{y-\mu}{\sigma}$ above 8.25; the function `Phi_approx` is more robust in the tails, but must be scaled and translated for anything other than a standard normal.

Available since 2.0

`real` **`normal_lcdf`**`(reals y | reals mu, reals sigma)`
The log of the cumulative normal distribution of y given location mu and scale sigma; normal_lcdf will underflow to $-\infty$ for $\frac{y-\mu}{\sigma}$ below -37.5 and overflow to 0 for $\frac{y-\mu}{\sigma}$ above 8.25; `log(Phi_approx(...))` is more robust in the tails, but must be scaled and translated for anything other than a standard normal.

Available since 2.12

`real` **`normal_lccdf`**`(reals y | reals mu, reals sigma)`
The log of the complementary cumulative normal distribution of y given location mu and scale sigma; normal_lccdf will overflow to 0 for $\frac{y-\mu}{\sigma}$ below -37.5 and underflow to $-\infty$ for $\frac{y-\mu}{\sigma}$ above 8.25; `log1m(Phi_approx(...))` is more robust in the tails, but must be scaled and translated for anything other than a standard normal.

Available since 2.15

`R` **`normal_rng`**`(reals mu, reals sigma)`
Generate a normal variate with location mu and scale sigma; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.18

Standard normal distribution

The standard normal distribution is so-called because its parameters are the units for their respective operations—the location (mean) is zero and the scale (standard deviation) one. The standard normal is parameter-free, and the unit parameters allow considerable simplification of the expression for the density.

$$\text{StdNormal}(y) = \text{Normal}(y \mid 0, 1) = \frac{1}{\sqrt{2\pi}} \exp\left(\frac{-y^2}{2}\right).$$

Up to a proportion on the log scale, where Stan computes,

$$\log \text{Normal}(y \mid 0, 1) = \frac{-y^2}{2} + \text{const}.$$

With no logarithm, no subtraction, and no division by a parameter, the standard normal log density is much more efficient to compute than the normal log density with constant location 0 and scale 1.

Distribution statement

`y ~ std_normal()`

Increment target log probability density with `std_normal_lupdf(y)`.

Available since 2.19

Stan functions

`real std_normal_lpdf(reals y)`
The standard normal (location zero, scale one) log probability density of y.

Available since 2.18

`real std_normal_lupdf(reals y)`
The standard normal (location zero, scale one) log probability density of y dropping constant additive terms.

Available since 2.25

`real std_normal_cdf(reals y)`
The cumulative standard normal distribution of y; std_normal_cdf will underflow to 0 for y below -37.5 and overflow to 1 for y above 8.25; the function `Phi_approx` is more robust in the tails.

Available since 2.21

`real std_normal_lcdf(reals y)`
The log of the cumulative standard normal distribution of y; std_normal_lcdf will underflow to $-\infty$ for y below -37.5 and overflow to 0 for y above 8.25; `log(Phi_approx(...))` is more robust in the tails.

Available since 2.21

`real std_normal_lccdf(reals y)`
The log of the complementary cumulative standard normal distribution of y; std_normal_lccdf will overflow to 0 for y below -37.5 and underflow to $-\infty$ for y above 8.25; `log1m(Phi_approx(...))` is more robust in the tails.

Available since 2.21

`R std_normal_qf(T x)`
Returns the value of the inverse standard normal cdf Φ^{-1} at the specified quantile `x`. The `std_normal_qf` is equivalent to the `inv_Phi` function.

Available since 2.31

`R` **`std_normal_log_qf`**`(T x)`
Return the value of the inverse standard normal cdf Φ^{-1} evaluated at the log of the specified quantile `x`. This function is equivalent to `std_normal_qf(exp(x))` but is more numerically stable.

Available since 2.31

`real` **`std_normal_rng`**`()`
Generate a normal variate with location zero and scale one; may only be used in transformed data and generated quantities blocks.

Available since 2.21

19.2. Normal-id generalized linear model (linear regression)

Stan also supplies a single function for a generalized linear model with normal distribution and identity link function, i.e. a function for a linear regression. This provides a more efficient implementation of linear regression than a manually written regression in terms of a normal distribution and matrix multiplication.

Probability distribution function

If $x \in \mathbb{R}^{n \cdot m}, \alpha \in \mathbb{R}^n, \beta \in \mathbb{R}^m, \sigma \in \mathbb{R}^+$, then for $y \in \mathbb{R}^n$,

$$\text{NormalIdGLM}(y|x,\alpha,\beta,\sigma) = \prod_{1 \le i \le n} \text{Normal}(y_i|\alpha_i + x_i \cdot \beta, \sigma).$$

Distribution statement

`y ~` **`normal_id_glm`**`(x, alpha, beta, sigma)`

Increment target log probability density with `normal_id_glm_lupdf(y | x, alpha, beta, sigma)`.

Available since 2.19

Stan functions

`real` **`normal_id_glm_lpdf`**`(real y | matrix x, real alpha, vector beta, real sigma)`
The log normal probability density of `y` given location `alpha + x * beta` and scale `sigma`.

Available since 2.29

`real` **`normal_id_glm_lupdf`**`(real y | matrix x, real alpha, vector beta, real sigma)`

The log normal probability density of `y` given location `alpha + x * beta` and scale `sigma` dropping constant additive terms.

Available since 2.29

`real` **`normal_id_glm_lpdf`**`(real y | matrix x, vector alpha, vector beta, real sigma)`
The log normal probability density of `y` given location `alpha + x * beta` and scale `sigma`.

Available since 2.29

`real` **`normal_id_glm_lupdf`**`(real y | matrix x, vector alpha, vector beta, real sigma)`
The log normal probability density of `y` given location `alpha + x * beta` and scale `sigma` dropping constant additive terms.

Available since 2.29

`real` **`normal_id_glm_lpdf`**`(real y | matrix x, real alpha, vector beta, vector sigma)`
The log normal probability density of `y` given location `alpha + x * beta` and scale `sigma`.

Available since 2.23

`real` **`normal_id_glm_lupdf`**`(real y | matrix x, real alpha, vector beta, vector sigma)`
The log normal probability density of `y` given location `alpha + x * beta` and scale `sigma` dropping constant additive terms.

Available since 2.25

`real` **`normal_id_glm_lpdf`**`(real y | matrix x, vector alpha, vector beta, vector sigma)`
The log normal probability density of `y` given location `alpha + x * beta` and scale `sigma`.

Available since 2.23

`real` **`normal_id_glm_lupdf`**`(real y | matrix x, vector alpha, vector beta, vector sigma)`
The log normal probability density of `y` given location `alpha + x * beta` and scale `sigma` dropping constant additive terms.

Available since 2.25

`real` **`normal_id_glm_lpdf`**`(vector y | row_vector x, real alpha, vector beta, real sigma)`
The log normal probability density of `y` given location `alpha + x * beta` and scale `sigma`.

Available since 2.29

`real` **`normal_id_glm_lupdf`**`(vector y | row_vector x, real alpha, vector beta, real sigma)`
The log normal probability density of `y` given location `alpha + x * beta` and scale `sigma` dropping constant additive terms.

Available since 2.29

`real` **`normal_id_glm_lpdf`**`(vector y | row_vector x, vector alpha, vector beta, real sigma)`
The log normal probability density of `y` given location `alpha + x * beta` and scale `sigma`.

Available since 2.29

`real` **`normal_id_glm_lupdf`**`(vector y | row_vector x, vector alpha, vector beta, real sigma)`
The log normal probability density of `y` given location `alpha + x * beta` and scale `sigma` dropping constant additive terms.

Available since 2.29

`real` **`normal_id_glm_lpdf`**`(vector y | matrix x, real alpha, vector beta, real sigma)`
The log normal probability density of `y` given location `alpha + x * beta` and scale `sigma`.

Available since 2.23

`real` **`normal_id_glm_lupdf`**`(vector y | matrix x, real alpha, vector beta, real sigma)`
The log normal probability density of `y` given location `alpha + x * beta` and scale `sigma` dropping constant additive terms.

Available since 2.23

`real` **`normal_id_glm_lpdf`**`(vector y | matrix x, vector alpha, vector beta, real sigma)`

The log normal probability density of `y` given location `alpha + x * beta` and scale `sigma`.

Available since 2.23

`real` **`normal_id_glm_lupdf`**`(vector y | matrix x, vector alpha, vector beta, real sigma)`
The log normal probability density of `y` given location `alpha + x * beta` and scale `sigma` dropping constant additive terms.

Available since 2.23

`real` **`normal_id_glm_lpdf`**`(vector y | matrix x, real alpha, vector beta, vector sigma)`
The log normal probability density of `y` given location `alpha + x * beta` and scale `sigma`.

Available since 2.30

`real` **`normal_id_glm_lupdf`**`(vector y | matrix x, real alpha, vector beta, vector sigma)`
The log normal probability density of `y` given location `alpha + x * beta` and scale `sigma` dropping constant additive terms.

Available since 2.30

`real` **`normal_id_glm_lpdf`**`(vector y | matrix x, vector alpha, vector beta, vector sigma)`
The log normal probability density of `y` given location `alpha + x * beta` and scale `sigma`.

Available since 2.30

`real` **`normal_id_glm_lupdf`**`(vector y | matrix x, vector alpha, vector beta, vector sigma)`
The log normal probability density of `y` given location `alpha + x * beta` and scale `sigma` dropping constant additive terms.

Available since 2.30

19.3. Exponentially modified normal distribution

Probability density function

If $\mu \in \mathbb{R}$, $\sigma \in \mathbb{R}^+$, and $\lambda \in \mathbb{R}^+$, then for $y \in \mathbb{R}$,

$$\text{ExpModNormal}(y|\mu,\sigma,\lambda) = \frac{\lambda}{2}\,\exp\left(\frac{\lambda}{2}\left(2\mu + \lambda\sigma^2 - 2y\right)\right)\text{erfc}\left(\frac{\mu + \lambda\sigma^2 - y}{\sqrt{2}\sigma}\right).$$

Distribution statement

`y ~ `**`exp_mod_normal`**`(mu, sigma, lambda)`

Increment target log probability density with `exp_mod_normal_lupdf(y | mu, sigma, lambda)`.

Available since 2.0

Stan functions

`real` **`exp_mod_normal_lpdf`**`(reals y | reals mu, reals sigma, reals lambda)`
The log of the exponentially modified normal density of y given location mu, scale sigma, and shape lambda

Available since 2.18

`real` **`exp_mod_normal_lupdf`**`(reals y | reals mu, reals sigma, reals lambda)`
The log of the exponentially modified normal density of y given location mu, scale sigma, and shape lambda dropping constant additive terms

Available since 2.25

`real` **`exp_mod_normal_cdf`**`(reals y | reals mu, reals sigma, reals lambda)`
The exponentially modified normal cumulative distribution function of y given location mu, scale sigma, and shape lambda

Available since 2.0

`real` **`exp_mod_normal_lcdf`**`(reals y | reals mu, reals sigma, reals lambda)`
The log of the exponentially modified normal cumulative distribution function of y given location mu, scale sigma, and shape lambda

Available since 2.18

real **exp_mod_normal_lccdf**(reals y | reals mu, reals sigma, reals lambda)
The log of the exponentially modified normal complementary cumulative distribution function of y given location mu, scale sigma, and shape lambda

Available since 2.18

R **exp_mod_normal_rng**(reals mu, reals sigma, reals lambda)
Generate a exponentially modified normal variate with location mu, scale sigma, and shape lambda; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.18

19.4. Skew normal distribution

Probability density function

If $\xi \in \mathbb{R}$, $\omega \in \mathbb{R}^+$, and $\alpha \in \mathbb{R}$, then for $y \in \mathbb{R}$,

$$\text{SkewNormal}(y \mid \xi, \omega, \alpha) = \frac{1}{\omega\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{y-\xi}{\omega}\right)^2\right) \left(1 + \text{erf}\left(\alpha\left(\frac{y-\xi}{\omega\sqrt{2}}\right)\right)\right).$$

Distribution statement

y ~ **skew_normal**(xi, omega, alpha)

Increment target log probability density with `skew_normal_lupdf(y | xi, omega, alpha)`.

Available since 2.0

Stan functions

real **skew_normal_lpdf**(reals y | reals xi, reals omega, reals alpha)
The log of the skew normal density of y given location xi, scale omega, and shape alpha

Available since 2.16

real **skew_normal_lupdf**(reals y | reals xi, reals omega, reals alpha)
The log of the skew normal density of y given location xi, scale omega, and shape alpha dropping constant additive terms

Available since 2.25

`real` **`skew_normal_cdf`**`(reals y | reals xi, reals omega, reals alpha)`
The skew normal distribution function of y given location xi, scale omega, and shape alpha

Available since 2.16

`real` **`skew_normal_lcdf`**`(reals y | reals xi, reals omega, reals alpha)`
The log of the skew normal cumulative distribution function of y given location xi, scale omega, and shape alpha

Available since 2.18

`real` **`skew_normal_lccdf`**`(reals y | reals xi, reals omega, reals alpha)`
The log of the skew normal complementary cumulative distribution function of y given location xi, scale omega, and shape alpha

Available since 2.18

`R` **`skew_normal_rng`**`(reals xi, reals omega, real alpha)`
Generate a skew normal variate with location xi, scale omega, and shape alpha; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.18

19.5. Student-t distribution

Probability density function

If $\nu \in \mathbb{R}^+$, $\mu \in \mathbb{R}$, and $\sigma \in \mathbb{R}^+$, then for $y \in \mathbb{R}$,

$$\text{StudentT}(y|\nu,\mu,\sigma) = \frac{\Gamma\left((\nu+1)/2\right)}{\Gamma(\nu/2)} \, \frac{1}{\sqrt{\nu\pi}\,\sigma} \, \left(1 + \frac{1}{\nu}\left(\frac{y-\mu}{\sigma}\right)^2\right)^{-(\nu+1)/2}.$$

Distribution statement

`y ~` **`student_t`**`(nu, mu, sigma)`

Increment target log probability density with `student_t_lupdf(y | nu, mu, sigma)`.

Available since 2.0

Stan functions

`real` **`student_t_lpdf`**`(reals y | reals nu, reals mu, reals sigma)`
The log of the Student-t density of y given degrees of freedom nu, location mu, and scale sigma

Available since 2.12

`real` **`student_t_lupdf`**`(reals y | reals nu, reals mu, reals sigma)`
The log of the Student-t density of y given degrees of freedom nu, location mu, and scale sigma dropping constant additive terms

Available since 2.25

`real` **`student_t_cdf`**`(reals y | reals nu, reals mu, reals sigma)`
The Student-t cumulative distribution function of y given degrees of freedom nu, location mu, and scale sigma

Available since 2.0

`real` **`student_t_lcdf`**`(reals y | reals nu, reals mu, reals sigma)`
The log of the Student-t cumulative distribution function of y given degrees of freedom nu, location mu, and scale sigma

Available since 2.12

`real` **`student_t_lccdf`**`(reals y | reals nu, reals mu, reals sigma)`
The log of the Student-t complementary cumulative distribution function of y given degrees of freedom nu, location mu, and scale sigma

Available since 2.12

`R` **`student_t_rng`**`(reals nu, reals mu, reals sigma)`
Generate a Student-t variate with degrees of freedom nu, location mu, and scale sigma; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.18

19.6. Cauchy distribution

Probability density function

If $\mu \in \mathbb{R}$ and $\sigma \in \mathbb{R}^+$, then for $y \in \mathbb{R}$,

$$\text{Cauchy}(y|\mu,\sigma) = \frac{1}{\pi\sigma} \frac{1}{1 + ((y-\mu)/\sigma)^2}.$$

Distribution statement

`y ~ cauchy(mu, sigma)`

Increment target log probability density with `cauchy_lupdf(y | mu, sigma)`.

Available since 2.0

Stan functions

`real cauchy_lpdf(reals y | reals mu, reals sigma)`
The log of the Cauchy density of y given location mu and scale sigma

Available since 2.12

`real cauchy_lupdf(reals y | reals mu, reals sigma)`
The log of the Cauchy density of y given location mu and scale sigma dropping constant additive terms

Available since 2.25

`real cauchy_cdf(reals y | reals mu, reals sigma)`
The Cauchy cumulative distribution function of y given location mu and scale sigma

Available since 2.0

`real cauchy_lcdf(reals y | reals mu, reals sigma)`
The log of the Cauchy cumulative distribution function of y given location mu and scale sigma

Available since 2.12

`real cauchy_lccdf(reals y | reals mu, reals sigma)`
The log of the Cauchy complementary cumulative distribution function of y given location mu and scale sigma

Available since 2.12

`R cauchy_rng(reals mu, reals sigma)`
Generate a Cauchy variate with location mu and scale sigma; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.18

19.7. Double exponential (Laplace) distribution

Probability density function

If $\mu \in \mathbb{R}$ and $\sigma \in \mathbb{R}^+$, then for $y \in \mathbb{R}$,

$$\text{DoubleExponential}(y|\mu,\sigma) = \frac{1}{2\sigma}\exp\left(-\frac{|y-\mu|}{\sigma}\right).$$

Note that the double exponential distribution is parameterized in terms of the scale, in contrast to the exponential distribution (see section exponential distribution), which is parameterized in terms of inverse scale.

The double-exponential distribution can be defined as a compound exponential-normal distribution (Ding and Blitzstein 2018). Using the inverse scale parameterization for the exponential distribution, and the standard deviation parameterization for the normal distribution, one can write

$$\alpha \sim \mathsf{Exponential}\left(\frac{1}{2\sigma^2}\right)$$

and

$$\beta \mid \alpha \sim \mathsf{Normal}(\mu, \sqrt{\alpha}),$$

then

$$\beta \sim \mathsf{DoubleExponential}(\mu, \sigma).$$

This may be used to code a non-centered parameterization by taking

$$\beta^{\text{raw}} \sim \mathsf{Normal}(0, 1)$$

and defining

$$\beta = \mu + \sqrt{\alpha}\,\beta^{\text{raw}}.$$

Distribution statement

`y ~ double_exponential(mu, sigma)`

Increment target log probability density with `double_exponential_lupdf(y | mu, sigma)`.

Available since 2.0

Stan functions

`real` **`double_exponential_lpdf`**`(reals y | reals mu, reals sigma)`
The log of the double exponential density of y given location mu and scale sigma

Available since 2.12

`real` **`double_exponential_lupdf`**`(reals y | reals mu, reals sigma)`
The log of the double exponential density of y given location mu and scale sigma dropping constant additive terms

Available since 2.25

`real` **`double_exponential_cdf`**`(reals y | reals mu, reals sigma)`
The double exponential cumulative distribution function of y given location mu and scale sigma

Available since 2.0

`real` **`double_exponential_lcdf`**`(reals y | reals mu, reals sigma)`
The log of the double exponential cumulative distribution function of y given location mu and scale sigma

Available since 2.12

`real` **`double_exponential_lccdf`**`(reals y | reals mu, reals sigma)`
The log of the double exponential complementary cumulative distribution function of y given location mu and scale sigma

Available since 2.12

`R` **`double_exponential_rng`**`(reals mu, reals sigma)`
Generate a double exponential variate with location mu and scale sigma; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.18

19.8. Logistic distribution

Probability density function

If $\mu \in \mathbb{R}$ and $\sigma \in \mathbb{R}^+$, then for $y \in \mathbb{R}$,

$$\text{Logistic}(y|\mu,\sigma) = \frac{1}{\sigma} \, \exp\!\left(-\frac{y-\mu}{\sigma}\right) \, \left(1 + \exp\!\left(-\frac{y-\mu}{\sigma}\right)\right)^{-2}.$$

Distribution statement

`y ~ logistic(mu, sigma)`

Increment target log probability density with `logistic_lupdf(y | mu, sigma)`.

Available since 2.0

Stan functions

`real logistic_lpdf(reals y | reals mu, reals sigma)`
The log of the logistic density of y given location mu and scale sigma

Available since 2.12

`real logistic_lupdf(reals y | reals mu, reals sigma)`
The log of the logistic density of y given location mu and scale sigma dropping constant additive terms

Available since 2.25

`real logistic_cdf(reals y | reals mu, reals sigma)`
The logistic cumulative distribution function of y given location mu and scale sigma

Available since 2.0

`real logistic_lcdf(reals y | reals mu, reals sigma)`
The log of the logistic cumulative distribution function of y given location mu and scale sigma

Available since 2.12

`real logistic_lccdf(reals y | reals mu, reals sigma)`
The log of the logistic complementary cumulative distribution function of y given location mu and scale sigma

Available since 2.12

`R logistic_rng(reals mu, reals sigma)`
Generate a logistic variate with location mu and scale sigma; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.18

19.9. Gumbel distribution

Probability density function

If $\mu \in \mathbb{R}$ and $\beta \in \mathbb{R}^+$, then for $y \in \mathbb{R}$,

$$\text{Gumbel}(y|\mu,\beta) = \frac{1}{\beta} \exp\left(-\frac{y-\mu}{\beta} - \exp\left(-\frac{y-\mu}{\beta}\right)\right).$$

Distribution statement

`y ~ `**`gumbel`**`(mu, beta)`

Increment target log probability density with `gumbel_lupdf(y | mu, beta)`.

Available since 2.0

Stan functions

`real` **`gumbel_lpdf`**`(reals y | reals mu, reals beta)`
The log of the gumbel density of y given location mu and scale beta

Available since 2.12

`real` **`gumbel_lupdf`**`(reals y | reals mu, reals beta)`
The log of the gumbel density of y given location mu and scale beta dropping constant additive terms

Available since 2.25

`real` **`gumbel_cdf`**`(reals y | reals mu, reals beta)`
The gumbel cumulative distribution function of y given location mu and scale beta

Available since 2.0

`real` **`gumbel_lcdf`**`(reals y | reals mu, reals beta)`
The log of the gumbel cumulative distribution function of y given location mu and scale beta

Available since 2.12

`real` **`gumbel_lccdf`**`(reals y | reals mu, reals beta)`
The log of the gumbel complementary cumulative distribution function of y given location mu and scale beta

Available since 2.12

`R` **`gumbel_rng`**`(reals mu, reals beta)`
Generate a gumbel variate with location mu and scale beta; may only be used in

transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.18

19.10. Skew double exponential distribution

Probability density function

If $\mu \in \mathbb{R}$, $\sigma \in \mathbb{R}^+$ and $\tau \in [0,1]$, then for $y \in \mathbb{R}$,

$$\text{SkewDoubleExponential}(y|\mu,\sigma,\tau) = \\ \frac{2\tau(1-\tau)}{\sigma} \exp\left[-\frac{2}{\sigma}\left[(1-\tau)\, I(y<\mu)(\mu-y) + \tau I(y>\mu)(y-\mu)\right]\right]$$

Distribution statement

`y ~ `**`skew_double_exponential`**`(mu, sigma, tau)`

Increment target log probability density with `skew_double_exponential(y | mu, sigma, tau)`

Available since 2.28

Stan functions

`real` **`skew_double_exponential_lpdf`**`(reals y | reals mu, reals sigma, reals tau)`
The log of the skew double exponential density of y given location mu, scale sigma and skewness tau

Available since 2.28

`real` **`skew_double_exponential_lupdf`**`(reals y | reals mu, reals sigma, reals tau)`
The log of the skew double exponential density of y given location mu, scale sigma and skewness tau dropping constant additive terms

Available since 2.28

`real` **`skew_double_exponential_cdf`**`(reals y | reals mu, reals sigma, reals tau)`
The skew double exponential cumulative distribution function of y given location mu, scale sigma and skewness tau

Available since 2.28

`real` **`skew_double_exponential_lcdf`**`(reals y | reals mu, reals sigma, reals tau)`
The log of the skew double exponential cumulative distribution function of y given location mu, scale sigma and skewness tau

Available since 2.28

`real` **`skew_double_exponential_lccdf`**`(reals y | reals mu, reals sigma, reals tau)`
The log of the skew double exponential complementary cumulative distribution function of y given location mu, scale sigma and skewness tau

Available since 2.28

`R` **`skew_double_exponential_rng`**`(reals mu, reals sigma, reals tau)`
Generate a skew double exponential variate with location mu, scale sigma and skewness tau; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.28

20. Positive Continuous Distributions

The positive continuous probability functions have support on the positive real numbers.

20.1. Lognormal distribution

Probability density function

If $\mu \in \mathbb{R}$ and $\sigma \in \mathbb{R}^+$, then for $y \in \mathbb{R}^+$,

$$\text{LogNormal}(y|\mu,\sigma) = \frac{1}{\sqrt{2\pi}\,\sigma}\,\frac{1}{y}\,\exp\left(-\frac{1}{2}\left(\frac{\log y - \mu}{\sigma}\right)^2\right).$$

Distribution statement

```
y ~ lognormal(mu, sigma)
```

Increment target log probability density with `lognormal_lupdf(y | mu, sigma)`.

Available since 2.0

Stan functions

`real lognormal_lpdf(reals y | reals mu, reals sigma)`
The log of the lognormal density of y given location mu and scale sigma

Available since 2.12

`real lognormal_lupdf(reals y | reals mu, reals sigma)`
The log of the lognormal density of y given location mu and scale sigma dropping constant additive terms

Available since 2.25

`real lognormal_cdf(reals y | reals mu, reals sigma)`
The cumulative lognormal distribution function of y given location mu and scale sigma

Available since 2.0

`real lognormal_lcdf(reals y | reals mu, reals sigma)`
The log of the lognormal cumulative distribution function of y given location mu and scale sigma

Available since 2.12

`real` **`lognormal_lccdf`**`(reals y | reals mu, reals sigma)`
The log of the lognormal complementary cumulative distribution function of y given location mu and scale sigma

Available since 2.12

`R` **`lognormal_rng`**`(reals mu, reals sigma)`
Generate a lognormal variate with location mu and scale sigma; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.22

20.2. Chi-square distribution

Probability density function

If $\nu \in \mathbb{R}^+$, then for $y \in \mathbb{R}^+$,

$$\text{ChiSquare}(y|\nu) = \frac{2^{-\nu/2}}{\Gamma(\nu/2)} \, y^{\nu/2-1} \exp\left(-\frac{1}{2} \, y\right).$$

Distribution statement

`y ~ ` **`chi_square`**`(nu)`

Increment target log probability density with `chi_square_lupdf(y | nu)`.

Available since 2.0

Stan functions

`real` **`chi_square_lpdf`**`(reals y | reals nu)`
The log of the Chi-square density of y given degrees of freedom nu

Available since 2.12

`real` **`chi_square_lupdf`**`(reals y | reals nu)`
The log of the Chi-square density of y given degrees of freedom nu dropping constant additive terms

Available since 2.25

`real` **`chi_square_cdf`**`(reals y | reals nu)`
The Chi-square cumulative distribution function of y given degrees of freedom nu

Available since 2.0

`real` **`chi_square_lcdf`**`(reals y | reals nu)`
The log of the Chi-square cumulative distribution function of y given degrees of freedom nu

Available since 2.12

`real` **`chi_square_lccdf`**`(reals y | reals nu)`
The log of the complementary Chi-square cumulative distribution function of y given degrees of freedom nu

Available since 2.12

`R` **`chi_square_rng`**`(reals nu)`
Generate a Chi-square variate with degrees of freedom nu; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.18

20.3. Inverse chi-square distribution

Probability density function

If $\nu \in \mathbb{R}^+$, then for $y \in \mathbb{R}^+$,

$$\text{InvChiSquare}(y \mid \nu) = \frac{2^{-\nu/2}}{\Gamma(\nu/2)}\, y^{-\nu/2-1} \exp\!\left(-\,\frac{1}{2}\,\frac{1}{y}\right).$$

Distribution statement

`y ~ `**`inv_chi_square`**`(nu)`

Increment target log probability density with `inv_chi_square_lupdf(y | nu)`.

Available since 2.0

Stan functions

`real` **`inv_chi_square_lpdf`**`(reals y | reals nu)`
The log of the inverse Chi-square density of y given degrees of freedom nu

Available since 2.12

`real` **`inv_chi_square_lupdf`**`(reals y | reals nu)`
The log of the inverse Chi-square density of y given degrees of freedom nu dropping constant additive terms

Available since 2.25

`real` **`inv_chi_square_cdf`**`(reals y | reals nu)`
The inverse Chi-squared cumulative distribution function of y given degrees of freedom nu

Available since 2.0

`real` **`inv_chi_square_lcdf`**`(reals y | reals nu)`
The log of the inverse Chi-squared cumulative distribution function of y given degrees of freedom nu

Available since 2.12

`real` **`inv_chi_square_lccdf`**`(reals y | reals nu)`
The log of the inverse Chi-squared complementary cumulative distribution function of y given degrees of freedom nu

Available since 2.12

`R` **`inv_chi_square_rng`**`(reals nu)`
Generate an inverse Chi-squared variate with degrees of freedom nu; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.18

20.4. Scaled inverse chi-square distribution

Probability density function

If $\nu \in \mathbb{R}^+$ and $\sigma \in \mathbb{R}^+$, then for $y \in \mathbb{R}^+$,

$$\text{ScaledInvChiSquare}(y|\nu,\sigma) = \frac{(\nu/2)^{\nu/2}}{\Gamma(\nu/2)}\, \sigma^{\nu}\, y^{-(\nu/2+1)}\, \exp\!\left(-\,\frac{1}{2}\,\nu\,\sigma^2\,\frac{1}{y}\right).$$

Distribution statement

`y ~` **`scaled_inv_chi_square`**`(nu, sigma)`

Increment target log probability density with `scaled_inv_chi_square_lupdf(y | nu, sigma)`.

Available since 2.0

Stan functions

`real` **`scaled_inv_chi_square_lpdf`**`(reals y | reals nu, reals sigma)`
The log of the scaled inverse Chi-square density of y given degrees of freedom nu and scale sigma

Available since 2.12

`real` **`scaled_inv_chi_square_lupdf`**`(reals y | reals nu, reals sigma)`
The log of the scaled inverse Chi-square density of y given degrees of freedom nu and scale sigma dropping constant additive terms

Available since 2.25

`real` **`scaled_inv_chi_square_cdf`**`(reals y | reals nu, reals sigma)`
The scaled inverse Chi-square cumulative distribution function of y given degrees of freedom nu and scale sigma

Available since 2.0

`real` **`scaled_inv_chi_square_lcdf`**`(reals y | reals nu, reals sigma)`
The log of the scaled inverse Chi-square cumulative distribution function of y given degrees of freedom nu and scale sigma

Available since 2.12

`real` **`scaled_inv_chi_square_lccdf`**`(reals y | reals nu, reals sigma)`
The log of the scaled inverse Chi-square complementary cumulative distribution function of y given degrees of freedom nu and scale sigma

Available since 2.12

`R` **`scaled_inv_chi_square_rng`**`(reals nu, reals sigma)`
Generate a scaled inverse Chi-squared variate with degrees of freedom nu and scale sigma; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.18

20.5. Exponential distribution

Probability density function

If $\beta \in \mathbb{R}^+$, then for $y \in \mathbb{R}^+$,

$$\text{Exponential}(y|\beta) = \beta \exp(-\beta\, y).$$

Distribution statement

`y ~` **`exponential`**`(beta)`

Increment target log probability density with `exponential_lupdf(y | beta)`.

Available since 2.0

Stan functions

`real` **`exponential_lpdf`**`(reals y | reals beta)`
The log of the exponential density of y given inverse scale beta

Available since 2.12

`real` **`exponential_lupdf`**`(reals y | reals beta)`
The log of the exponential density of y given inverse scale beta dropping constant additive terms

Available since 2.25

`real` **`exponential_cdf`**`(reals y | reals beta)`
The exponential cumulative distribution function of y given inverse scale beta

Available since 2.0

`real` **`exponential_lcdf`**`(reals y | reals beta)`
The log of the exponential cumulative distribution function of y given inverse scale beta

Available since 2.12

`real` **`exponential_lccdf`**`(reals y | reals beta)`
The log of the exponential complementary cumulative distribution function of y given inverse scale beta

Available since 2.12

`R` **`exponential_rng`**`(reals beta)`
Generate an exponential variate with inverse scale beta; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.18

20.6. Gamma distribution

Probability density function

If $\alpha \in \mathbb{R}^+$ and $\beta \in \mathbb{R}^+$, then for $y \in \mathbb{R}^+$,

$$\text{Gamma}(y|\alpha,\beta) = \frac{\beta^\alpha}{\Gamma(\alpha)} \, y^{\alpha-1} \exp(-\beta \, y).$$

Distribution statement

`y ~` **`gamma`**`(alpha, beta)`

Increment target log probability density with `gamma_lupdf(y | alpha, beta)`.

Available since 2.0

Stan functions

`real` **`gamma_lpdf`**`(reals y | reals alpha, reals beta)`
The log of the gamma density of y given shape alpha and inverse scale beta

Available since 2.12

`real` **`gamma_lupdf`**`(reals y | reals alpha, reals beta)`
The log of the gamma density of y given shape alpha and inverse scale beta dropping constant additive terms

Available since 2.25

`real` **`gamma_cdf`**`(reals y | reals alpha, reals beta)`
The cumulative gamma distribution function of y given shape alpha and inverse scale beta

Available since 2.0

`real` **`gamma_lcdf`**`(reals y | reals alpha, reals beta)`
The log of the cumulative gamma distribution function of y given shape alpha and inverse scale beta

Available since 2.12

`real` **`gamma_lccdf`**`(reals y | reals alpha, reals beta)`
The log of the complementary cumulative gamma distribution function of y given shape alpha and inverse scale beta

Available since 2.12

`R` **`gamma_rng`**`(reals alpha, reals beta)`
Generate a gamma variate with shape alpha and inverse scale beta; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.18

20.7. Inverse gamma Distribution

Probability density function

If $\alpha \in \mathbb{R}^+$ and $\beta \in \mathbb{R}^+$, then for $y \in \mathbb{R}^+$,

$$\text{InvGamma}(y|\alpha,\beta) = \frac{\beta^\alpha}{\Gamma(\alpha)} \, y^{-(\alpha+1)} \, \exp\left(-\beta \, \frac{1}{y}\right).$$

Distribution statement

`y ~ `**`inv_gamma`**`(alpha, beta)`

Increment target log probability density with `inv_gamma_lupdf(y | alpha, beta)`.

Available since 2.0

Stan functions

`real `**`inv_gamma_lpdf`**`(reals y | reals alpha, reals beta)`
The log of the inverse gamma density of y given shape alpha and scale beta

Available since 2.12

`real `**`inv_gamma_lupdf`**`(reals y | reals alpha, reals beta)`
The log of the inverse gamma density of y given shape alpha and scale beta dropping constant additive terms

Available since 2.25

`real `**`inv_gamma_cdf`**`(reals y | reals alpha, reals beta)`
The inverse gamma cumulative distribution function of y given shape alpha and scale beta

Available since 2.0

`real `**`inv_gamma_lcdf`**`(reals y | reals alpha, reals beta)`
The log of the inverse gamma cumulative distribution function of y given shape alpha and scale beta

Available since 2.12

`real `**`inv_gamma_lccdf`**`(reals y | reals alpha, reals beta)`
The log of the inverse gamma complementary cumulative distribution function of y given shape alpha and scale beta

Available since 2.12

`R` **`inv_gamma_rng`**`(reals alpha, reals beta)`
Generate an inverse gamma variate with shape alpha and scale beta; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.18

20.8. Weibull distribution

Probability density function

If $\alpha \in \mathbb{R}^+$ and $\sigma \in \mathbb{R}^+$, then for $y \in [0, \infty)$,

$$\text{Weibull}(y|\alpha,\sigma) = \frac{\alpha}{\sigma} \left(\frac{y}{\sigma}\right)^{\alpha-1} \exp\left(-\left(\frac{y}{\sigma}\right)^{\alpha}\right).$$

Note that if $Y \propto \text{Weibull}(\alpha, \sigma)$, then $Y^{-1} \propto \text{Frechet}(\alpha, \sigma^{-1})$.

Distribution statement

`y ~` **`weibull`**`(alpha, sigma)`

Increment target log probability density with `weibull_lupdf(y | alpha, sigma)`.

Available since 2.0

Stan functions

`real` **`weibull_lpdf`**`(reals y | reals alpha, reals sigma)`
The log of the Weibull density of y given shape alpha and scale sigma

Available since 2.12

`real` **`weibull_lupdf`**`(reals y | reals alpha, reals sigma)`
The log of the Weibull density of y given shape alpha and scale sigma dropping constant additive terms

Available since 2.25

`real` **`weibull_cdf`**`(reals y | reals alpha, reals sigma)`
The Weibull cumulative distribution function of y given shape alpha and scale sigma

Available since 2.0

`real` **`weibull_lcdf`**`(reals y | reals alpha, reals sigma)`
The log of the Weibull cumulative distribution function of y given shape alpha and scale sigma

Available since 2.12

`real` **`weibull_lccdf`**`(reals y | reals alpha, reals sigma)`
The log of the Weibull complementary cumulative distribution function of y given shape alpha and scale sigma

Available since 2.12

`R` **`weibull_rng`**`(reals alpha, reals sigma)`
Generate a weibull variate with shape alpha and scale sigma; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.18

20.9. Frechet distribution

Probability density function

If $\alpha \in \mathbb{R}^+$ and $\sigma \in \mathbb{R}^+$, then for $y \in \mathbb{R}^+$,

$$\text{Frechet}(y|\alpha,\sigma) = \frac{\alpha}{\sigma} \left(\frac{y}{\sigma}\right)^{-\alpha-1} \exp\left(-\left(\frac{y}{\sigma}\right)^{-\alpha}\right).$$

Note that if $Y \propto \text{Frechet}(\alpha,\sigma)$, then $Y^{-1} \propto \text{Weibull}(\alpha,\sigma^{-1})$.

Distribution statement

`y ~ `**`frechet`**`(alpha, sigma)`

Increment target log probability density with `frechet_lupdf(y | alpha, sigma)`.

Available since 2.5

Stan functions

`real` **`frechet_lpdf`**`(reals y | reals alpha, reals sigma)`
The log of the Frechet density of y given shape alpha and scale sigma

Available since 2.12

`real` **`frechet_lupdf`**`(reals y | reals alpha, reals sigma)`
The log of the Frechet density of y given shape alpha and scale sigma dropping constant additive terms

Available since 2.25

real **frechet_cdf**(reals y | reals alpha, reals sigma)
The Frechet cumulative distribution function of y given shape alpha and scale sigma

Available since 2.5

real **frechet_lcdf**(reals y | reals alpha, reals sigma)
The log of the Frechet cumulative distribution function of y given shape alpha and scale sigma

Available since 2.12

real **frechet_lccdf**(reals y | reals alpha, reals sigma)
The log of the Frechet complementary cumulative distribution function of y given shape alpha and scale sigma

Available since 2.12

R **frechet_rng**(reals alpha, reals sigma)
Generate a Frechet variate with shape alpha and scale sigma; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.18

20.10. Rayleigh distribution

Probability density function

If $\sigma \in \mathbb{R}^+$, then for $y \in [0, \infty)$,

$$\text{Rayleigh}(y|\sigma) = \frac{y}{\sigma^2} \exp(-y^2/2\sigma^2).$$

Distribution statement

y ~ **rayleigh**(sigma)

Increment target log probability density with `rayleigh_lupdf(y | sigma)`.

Available since 2.0

Stan functions

real **rayleigh_lpdf**(reals y | reals sigma)
The log of the Rayleigh density of y given scale sigma

Available since 2.12

`real` **`rayleigh_lupdf`**`(reals y | reals sigma)`
The log of the Rayleigh density of y given scale sigma dropping constant additive terms

Available since 2.25

`real` **`rayleigh_cdf`**`(real y | real sigma)`
The Rayleigh cumulative distribution of y given scale sigma

Available since 2.0

`real` **`rayleigh_lcdf`**`(real y | real sigma)`
The log of the Rayleigh cumulative distribution of y given scale sigma

Available since 2.12

`real` **`rayleigh_lccdf`**`(real y | real sigma)`
The log of the Rayleigh complementary cumulative distribution of y given scale sigma

Available since 2.12

`R` **`rayleigh_rng`**`(reals sigma)`
Generate a Rayleigh variate with scale sigma; may only be used in generated quantities block. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.18

20.11. Log-logistic distribution

Probability density function

If $\alpha, \beta \in \mathbb{R}^+$, then for $y \in \mathbb{R}^+$,

$$\text{Log-Logistic}(y|\alpha,\beta) = \frac{\left(\frac{\beta}{\alpha}\right)\left(\frac{y}{\alpha}\right)^{\beta-1}}{\left(1+\left(\frac{y}{\alpha}\right)^{\beta}\right)^2}.$$

Distribution statement

`y ~` **`loglogistic`**`(alpha, beta)`

Increment target log probability density with unnormalized version of `loglogistic_lpdf(y | alpha, beta)`

Available since 2.29

Stan functions

`real` **`loglogistic_lpdf`**`(reals y | reals alpha, reals beta)`
The log of the log-logistic density of y given scale alpha and shape beta

Available since 2.29

`real` **`loglogistic_cdf`**`(reals y | reals alpha, reals beta)`
The log-logistic cumulative distribution function of y given scale alpha and shape beta

Available since 2.29

`R` **`loglogistic_rng`**`(reals alpha, reals beta)`
Generate a log-logistic variate with scale alpha and shape beta; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.29

21. Positive Lower-Bounded Distributions

The positive lower-bounded probabilities have support on real values above some positive minimum value.

21.1. Pareto distribution

Probability density function

If $y_{\text{min}} \in \mathbb{R}^+$ and $\alpha \in \mathbb{R}^+$, then for $y \in \mathbb{R}^+$ with $y \geq y_{\text{min}}$,

$$\text{Pareto}(y|y_{\text{min}}, \alpha) = \frac{\alpha\, y_{\text{min}}^\alpha}{y^{\alpha+1}}.$$

Distribution statement

```
y ~ pareto(y_min, alpha)
```

Increment target log probability density with `pareto_lupdf(y | y_min, alpha)`.

Available since 2.0

Stan functions

`real` **`pareto_lpdf`**`(reals y | reals y_min, reals alpha)`
The log of the Pareto density of y given positive minimum value y_min and shape alpha

Available since 2.12

`real` **`pareto_lupdf`**`(reals y | reals y_min, reals alpha)`
The log of the Pareto density of y given positive minimum value y_min and shape alpha dropping constant additive terms

Available since 2.25

`real` **`pareto_cdf`**`(reals y | reals y_min, reals alpha)`
The Pareto cumulative distribution function of y given positive minimum value y_min and shape alpha

Available since 2.0

`real` **`pareto_lcdf`**`(reals y | reals y_min, reals alpha)`
The log of the Pareto cumulative distribution function of y given positive minimum value y_min and shape alpha

Available since 2.12

`real` **`pareto_lccdf`**`(reals y | reals y_min, reals alpha)`
The log of the Pareto complementary cumulative distribution function of y given positive minimum value y_min and shape alpha

Available since 2.12

`R` **`pareto_rng`**`(reals y_min, reals alpha)`
Generate a Pareto variate with positive minimum value y_min and shape alpha; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.18

21.2. Pareto type 2 distribution

Probability density function

If $\mu \in \mathbb{R}$, $\lambda \in \mathbb{R}^+$, and $\alpha \in \mathbb{R}^+$, then for $y \geq \mu$,

$$\text{Pareto\_Type\_2}(y|\mu, \lambda, \alpha) = \frac{\alpha}{\lambda} \left(1 + \frac{y - \mu}{\lambda}\right)^{-(\alpha+1)}.$$

Note that the Lomax distribution is a Pareto Type 2 distribution with $\mu = 0$.

Distribution statement

`y ~` **`pareto_type_2`**`(mu, lambda, alpha)`

Increment target log probability density with `pareto_type_2_lupdf(y | mu, lambda, alpha)`.

Available since 2.5

Stan functions

`real` **`pareto_type_2_lpdf`**`(reals y | reals mu, reals lambda, reals alpha)`
The log of the Pareto Type 2 density of y given location mu, scale lambda, and shape alpha

Available since 2.18

`real` **`pareto_type_2_lupdf`**`(reals y | reals mu, reals lambda, reals alpha)`
The log of the Pareto Type 2 density of y given location mu, scale lambda, and shape alpha dropping constant additive terms

Available since 2.25

`real` **`pareto_type_2_cdf`**`(reals y | reals mu, reals lambda, reals alpha)`
The Pareto Type 2 cumulative distribution function of y given location mu, scale lambda, and shape alpha

Available since 2.5

`real` **`pareto_type_2_lcdf`**`(reals y | reals mu, reals lambda, reals alpha)`
The log of the Pareto Type 2 cumulative distribution function of y given location mu, scale lambda, and shape alpha

Available since 2.18

`real` **`pareto_type_2_lccdf`**`(reals y | reals mu, reals lambda, reals alpha)`
The log of the Pareto Type 2 complementary cumulative distribution function of y given location mu, scale lambda, and shape alpha

Available since 2.18

`R` **`pareto_type_2_rng`**`(reals mu, reals lambda, reals alpha)`
Generate a Pareto Type 2 variate with location mu, scale lambda, and shape alpha; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.18

21.3. Wiener First Passage Time Distribution

Probability density function

If $\alpha \in \mathbb{R}^+$, $\tau \in \mathbb{R}^+$, $\beta \in (0,1)$, $\delta \in \mathbb{R}$, $s_\delta \in \mathbb{R}^{\geq 0}$, $s_\beta \in [0,1)$, and $s_\tau \in \mathbb{R}^{\geq 0}$ then for $y > \tau$,

$$\begin{aligned}
&\text{Wiener}(y \mid \alpha, \tau, \beta, \delta, s_\delta, s_\beta, s_\tau) = \\
&\frac{1}{s_\tau} \int_\tau^{\tau+s_\tau} \frac{1}{s_\beta} \int_{\beta-\frac{1}{2}s_\beta}^{\beta+\frac{1}{2}s_\beta} \int_{-\infty}^{\infty} p_3(y-\tau_0 \mid \alpha, \nu, \omega) \\
&\times \frac{1}{\sqrt{2\pi s_\delta^2}} \exp\Big(-\frac{(\nu-\delta)^2}{2s_\delta^2}\Big)\, d\nu\, d\omega\, d\tau_0 = \\
&\frac{1}{s_\tau} \int_\tau^{\tau+s_\tau} \frac{1}{s_\beta} \int_{\beta-\frac{1}{2}s_\beta}^{\beta+\frac{1}{2}s_\beta} M \times p_3(y-\tau_0 \mid \alpha, \nu, \omega)\, d\omega\, d\tau_0,
\end{aligned}$$

where $p()$ denotes the density function, and M and $p_3()$ are defined, by using $t := y - \tau_0$, as

$$M := \frac{1}{\sqrt{1+s_\delta^2 t}} \exp\Big(\alpha\delta\omega + \frac{\delta^2 t}{2} + \frac{s_\delta^2\alpha^2\omega^2 - 2\alpha\delta\omega - \delta^2 t}{2(1+s_\delta^2 t)}\Big) \text{ and}$$

$$p_3(t \mid \alpha, \delta, \beta) := \frac{1}{\alpha^2} \exp\Big(-\alpha\delta\beta - \frac{\delta^2 t}{2}\Big) f\Big(\frac{t}{\alpha^2} \mid 0, 1, \beta\Big),$$

where $f(t^* = \frac{t}{\alpha^2} \mid 0, 1, \beta)$ can be specified in two ways:

$$f_l(t^* \mid 0, 1, \beta) = \sum_{k=1}^{\infty} k\pi \exp\Big(-\frac{k^2\pi^2 t^*}{2}\Big) \sin(k\pi\beta) \text{ and}$$

$$f_s(t^* \mid 0, 1, \beta) = \sum_{k=-\infty}^{\infty} \frac{1}{\sqrt{2\pi(t^*)^3}} (\beta + 2k) \exp\Big(-\frac{(\beta+2k)^2}{2t^*}\Big).$$

Which of these is used in the computations depends on which expression requires the smaller number of components k to guarantee a pre-specified precision

In the case where s_δ, s_β, and s_τ are all 0, this simplifies to

$$\text{Wiener}(y|\alpha, \tau, \beta, \delta) = \frac{\alpha^3}{(y-\tau)^{3/2}} \exp\Big(-\delta\alpha\beta - \frac{\delta^2(y-\tau)}{2}\Big) \sum_{k=-\infty}^{\infty} (2k+\beta)\phi\Big(\frac{2k\alpha+\beta}{\sqrt{y-\tau}}\Big)$$

where $\phi(x)$ denotes the standard normal density function; see (Feller 1968), (Navarro and Fuss 2009).

Distribution statement

`y ~ wiener(alpha, tau, beta, delta)`

Increment target log probability density with `wiener_lupdf(y | alpha, tau, beta, delta)`.

Available since 2.7

`y ~ wiener(alpha, tau, beta, delta, var_delta)` Increment target log probability density with `wiener_lupdf(y | alpha, tau, beta, delta, var_delta)`.

Available since 2.35

`y ~ wiener(alpha, tau, beta, delta, var_delta, var_beta, var_tau)` Increment target log probability density with `wiener_lupdf(y | alpha, tau, beta, delta, var_delta, var_beta, var_tau)`.

Available since 2.35

Stan functions

`real wiener_lpdf(reals y | reals alpha, reals tau, reals beta, reals delta)`
The log of the Wiener first passage time density of `y` given boundary separation `alpha`, non-decision time `tau`, a-priori bias `beta`, and drift rate `delta`.

Available since 2.18

`real wiener_lpdf(real y | real alpha, real tau, real beta, real delta, real var_delta)`
The log of the Wiener first passage time density of `y` given boundary separation `alpha`, non-decision time `tau`, a-priori bias `beta`, drift rate `delta`, and inter-trial drift rate variability `var_delta`.

Setting `var_delta` to `0` recovers the 4-parameter signature above.

Available since 2.35

`real wiener_lpdf(real y | real alpha, real tau, real beta, real delta, real var_delta, real var_beta, real var_tau)`
The log of the Wiener first passage time density of `y` given boundary separation `alpha`, non-decision time `tau`, a-priori bias `beta`, drift rate `delta`, inter-trial drift rate variability `var_delta`, inter-trial variability of the starting point (bias) `var_beta`, and inter-trial variability of the non-decision time `var_tau`.

Setting `var_delta`, `var_beta`, and `var_tau` to `0` recovers the 4-parameter signature above.

Available since 2.35

`real` **`wiener_lupdf`**`(reals y | reals alpha, reals tau, reals beta, reals delta)`
The log of the Wiener first passage time density of `y` given boundary separation `alpha`, non-decision time `tau`, a-priori bias `beta`, and drift rate `delta`, dropping constant additive terms

Available since 2.25

`real` **`wiener_lupdf`**`(real y | real alpha, real tau, real beta, real delta, real var_delta)`
The log of the Wiener first passage time density of `y` given boundary separation `alpha`, non-decision time `tau`, a-priori bias `beta`, drift rate `delta`, and inter-trial drift rate variability `var_delta`, dropping constant additive terms.

Setting `var_delta` to `0` recovers the 4-parameter signature above.

Available since 2.35

`real` **`wiener_lupdf`**`(real y | real alpha, real tau, real beta, real delta, real var_delta, real var_beta, real var_tau)`
The log of the Wiener first passage time density of `y` given boundary separation `alpha`, non-decision time `tau`, a-priori bias `beta`, drift rate `delta`, inter-trial drift rate variability `var_delta`, inter-trial variability of the starting point (bias) `var_beta`, and inter-trial variability of the non-decision time `var_tau`, dropping constant additive terms.

Setting `var_delta`, `var_beta`, and `var_tau` to `0` recovers the 4-parameter signature above.

Available since 2.35

Boundaries

Stan returns the first passage time of the accumulation process over the upper boundary only. To get the result for the lower boundary, use

$$\text{Wiener}(y|\alpha, \tau, 1 - \beta, -\delta)$$

For more details, see the appendix of Vandekerckhove and Wabersich (2014).

22. Continuous Distributions on [0, 1]

The continuous distributions with outcomes in the interval $[0,1]$ are used to characterized bounded quantities, including probabilities.

22.1. Beta distribution

Probability density function

If $\alpha \in \mathbb{R}^+$ and $\beta \in \mathbb{R}^+$, then for $\theta \in (0,1)$,

$$\text{Beta}(\theta|\alpha,\beta) = \frac{1}{\mathrm{B}(\alpha,\beta)}\,\theta^{\alpha-1}\,(1-\theta)^{\beta-1},$$

where the beta function $\mathrm{B}()$ is as defined in section combinatorial functions.

Warning: If $\theta = 0$ or $\theta = 1$, then the probability is 0 and the log probability is $-\infty$. Similarly, the distribution requires strictly positive parameters, $\alpha, \beta > 0$.

Distribution statement

`theta ~ beta(alpha, beta)`

Increment target log probability density with `beta_lupdf(theta | alpha, beta)`.

Available since 2.0

Stan functions

`real beta_lpdf(reals theta | reals alpha, reals beta)`
The log of the beta density of `theta` in $[0,1]$ given positive prior successes (plus one) alpha and prior failures (plus one) beta

Available since 2.12

`real beta_lupdf(reals theta | reals alpha, reals beta)`
The log of the beta density of `theta` in $[0,1]$ given positive prior successes (plus one) alpha and prior failures (plus one) beta dropping constant additive terms

Available since 2.25

`real beta_cdf(reals theta | reals alpha, reals beta)`
The beta cumulative distribution function of `theta` in $[0,1]$ given positive prior successes (plus one) alpha and prior failures (plus one) beta

Available since 2.0

`real` **`beta_lcdf`**`(reals theta | reals alpha, reals beta)`
The log of the beta cumulative distribution function of `theta` in $[0, 1]$ given positive prior successes (plus one) alpha and prior failures (plus one) beta

Available since 2.12

`real` **`beta_lccdf`**`(reals theta | reals alpha, reals beta)`
The log of the beta complementary cumulative distribution function of `theta` in $[0, 1]$ given positive prior successes (plus one) alpha and prior failures (plus one) beta

Available since 2.12

`R` **`beta_rng`**`(reals alpha, reals beta)`
Generate a beta variate with positive prior successes (plus one) alpha and prior failures (plus one) beta; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.18

22.2. Beta proportion distribution

Probability density function

If $\mu \in (0, 1)$ and $\kappa \in \mathbb{R}^+$, then for $\theta \in (0, 1)$,

$$\text{Beta\_Proportion}(\theta|\mu, \kappa) = \frac{1}{\mathrm{B}(\mu\kappa, (1-\mu)\kappa)} \, \theta^{\mu\kappa-1} \, (1-\theta)^{(1-\mu)\kappa-1},$$

where the beta function $\mathrm{B}()$ is as defined in section combinatorial functions.

Warning: If $\theta = 0$ or $\theta = 1$, then the probability is 0 and the log probability is $-\infty$. Similarly, the distribution requires $\mu \in (0, 1)$ and strictly positive parameter, $\kappa > 0$.

Distribution statement

`theta ~` **`beta_proportion`**`(mu, kappa)`

Increment target log probability density with `beta_proportion_lupdf(theta | mu, kappa)`.

Available since 2.19

Stan functions

`real` **`beta_proportion_lpdf`**`(reals theta | reals mu, reals kappa)`
The log of the beta_proportion density of `theta` in $(0, 1)$ given mean mu and precision kappa

Available since 2.19

`real` **`beta_proportion_lupdf`**`(reals theta | reals mu, reals kappa)`
The log of the beta_proportion density of `theta` in $(0, 1)$ given mean mu and precision kappa dropping constant additive terms

Available since 2.25

`real` **`beta_proportion_lcdf`**`(reals theta | reals mu, reals kappa)`
The log of the beta_proportion cumulative distribution function of `theta` in $(0, 1)$ given mean mu and precision kappa

Available since 2.18

`real` **`beta_proportion_lccdf`**`(reals theta | reals mu, reals kappa)`
The log of the beta_proportion complementary cumulative distribution function of `theta` in $(0, 1)$ given mean mu and precision kappa

Available since 2.18

`R` **`beta_proportion_rng`**`(reals mu, reals kappa)`
Generate a beta_proportion variate with mean mu and precision kappa; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.18

23. Circular Distributions

Circular distributions are defined for finite values y in any interval of length 2π.

23.1. Von Mises distribution

Probability density function

If $\mu \in \mathbb{R}$ and $\kappa \in \mathbb{R}^+$, then for $y \in \mathbb{R}$,

$$\text{VonMises}(y|\mu, \kappa) = \frac{\exp(\kappa \cos(y - \mu))}{2\pi I_0(\kappa)}.$$

In order for this density to properly normalize, y must be restricted to some interval $(c, c + 2\pi)$ of length 2π, because

$$\int_c^{c+2\pi} \text{VonMises}(y|\mu, \kappa) dy = 1.$$

Similarly, if μ is a parameter, it will typically be restricted to the same range as y.

If $\kappa > 0$, a von Mises distribution with its 2π interval of support centered around its location μ will have a single mode at μ; for example, restricting y to $(-\pi, \pi)$ and taking $\mu = 0$ leads to a single local optimum at the mode μ. If the location μ is not in the center of the support, the density is circularly translated and there will be a second local maximum at the boundary furthest from the mode. Ideally, the parameterization and support will be set up so that the bulk of the probability mass is in a continuous interval around the mean μ.

For $\kappa = 0$, the Von Mises distribution corresponds to the circular uniform distribution with density $1/(2\pi)$ (independently of the values of y or μ).

Distribution statement

```
y ~ von_mises(mu, kappa)
```

Increment target log probability density with `von_mises_lupdf(y | mu, kappa)`.

Available since 2.0

Stan functions

`real` **`von_mises_lpdf`**`(reals y | reals mu, reals kappa)`
The log of the von mises density of y given location mu and scale kappa.

Available since 2.18

`real` **`von_mises_lupdf`**`(reals y | reals mu, reals kappa)`
The log of the von mises density of y given location mu and scale kappa dropping constant additive terms.

Available since 2.25

`real` **`von_mises_cdf`**`(reals y | reals mu, reals kappa)`
The von mises cumulative distribution function of y given location mu and scale kappa.

Available since 2.29

`real` **`von_mises_lcdf`**`(reals y | reals mu, reals kappa)`
The log of the von mises cumulative distribution function of y given location mu and scale kappa.

Available since 2.29

`real` **`von_mises_lccdf`**`(reals y | reals mu, reals kappa)`
The log of the von mises complementary cumulative distribution function of y given location mu and scale kappa.

Available since 2.29

`R` **`von_mises_rng`**`(reals mu, reals kappa)`
Generate a Von Mises variate with location mu and scale kappa (i.e. returns values in the interval $[(\mu \mod 2\pi) - \pi, (\mu \mod 2\pi) + \pi]$); may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.18

Numerical stability

Evaluating the Von Mises distribution for $\kappa > 100$ is numerically unstable in the current implementation. Nathanael I. Lichti suggested the following workaround on the Stan users group, based on the fact that as $\kappa \to \infty$,

$$\text{VonMises}(y|\mu, \kappa) \to \text{Normal}(\mu, \sqrt{1/\kappa}).$$

The workaround is to replace `y ~ von_mises(mu,kappa)` with

```
if (kappa < 100) {
  y ~ von_mises(mu, kappa);
} else {
```

```
  y ~ normal(mu, sqrt(1 / kappa));
}
```

24. Bounded Continuous Distributions

The bounded continuous probabilities have support on a finite interval of real numbers.

24.1. Uniform distribution

Probability density function

If $\alpha \in \mathbb{R}$ and $\beta \in (\alpha, \infty)$, then for $y \in [\alpha, \beta]$,

$$\text{Uniform}(y|\alpha, \beta) = \frac{1}{\beta - \alpha}.$$

Distribution statement

`y ~ uniform(alpha, beta)`

Increment target log probability density with `uniform_lupdf(y | alpha, beta)`.

Available since 2.0

Stan functions

`real uniform_lpdf(reals y | reals alpha, reals beta)`
The log of the uniform density of y given lower bound alpha and upper bound beta

Available since 2.12

`real uniform_lupdf(reals y | reals alpha, reals beta)`
The log of the uniform density of y given lower bound alpha and upper bound beta dropping constant additive terms

Available since 2.25

`real uniform_cdf(reals y | reals alpha, reals beta)`
The uniform cumulative distribution function of y given lower bound alpha and upper bound beta

Available since 2.0

`real uniform_lcdf(reals y | reals alpha, reals beta)`
The log of the uniform cumulative distribution function of y given lower bound alpha and upper bound beta

Available since 2.12

`real` **`uniform_lccdf`**`(reals y | reals alpha, reals beta)`
The log of the uniform complementary cumulative distribution function of y given lower bound alpha and upper bound beta

Available since 2.12

`R` **`uniform_rng`**`(reals alpha, reals beta)`
Generate a uniform variate with lower bound alpha and upper bound beta; may only be used in transformed data and generated quantities blocks. For a description of argument and return types, see section vectorized PRNG functions.

Available since 2.18

25. Distributions over Unbounded Vectors

The unbounded vector probability distributions have support on all of $\mathbb{R}^K$ for some fixed K.

25.1. Multivariate normal distribution

Probability density function

If $K \in \mathbb{N}$, $\mu \in \mathbb{R}^K$, and $\Sigma \in \mathbb{R}^{K \times K}$ is symmetric and positive definite, then for $y \in \mathbb{R}^K$,

$$\text{MultiNormal}(y|\mu,\Sigma) = \frac{1}{(2\pi)^{K/2}} \frac{1}{\sqrt{|\Sigma|}} \exp\left(-\frac{1}{2}(y-\mu)^\top \Sigma^{-1} (y-\mu)\right),$$

where $|\Sigma|$ is the absolute determinant of Σ.

Distribution statement

`y ~ `**`multi_normal`**`(mu, Sigma)`

Increment target log probability density with `multi_normal_lupdf(y | mu, Sigma)`.

Available since 2.0

Stan functions

The multivariate normal probability function is overloaded to allow the variate vector y and location vector μ to be vectors or row vectors (or to mix the two types). The density function is also vectorized, so it allows arrays of row vectors or vectors as arguments; see section vectorized function signatures for a description of vectorization.

`real `**`multi_normal_lpdf`**`(vectors y | vectors mu, matrix Sigma)`
The log of the multivariate normal density of vector(s) y given location vector(s) mu and covariance matrix Sigma

Available since 2.12

`real `**`multi_normal_lupdf`**`(vectors y | vectors mu, matrix Sigma)`
The log of the multivariate normal density of vector(s) y given location vector(s) mu and covariance matrix Sigma dropping constant additive terms

Available since 2.25

`real` **`multi_normal_lpdf`**`(vectors y | row_vectors mu, matrix Sigma)`
The log of the multivariate normal density of vector(s) y given location row vector(s) mu and covariance matrix Sigma

Available since 2.12

`real` **`multi_normal_lupdf`**`(vectors y | row_vectors mu, matrix Sigma)`
The log of the multivariate normal density of vector(s) y given location row vector(s) mu and covariance matrix Sigma dropping constant additive terms

Available since 2.25

`real` **`multi_normal_lpdf`**`(row_vectors y | vectors mu, matrix Sigma)`
The log of the multivariate normal density of row vector(s) y given location vector(s) mu and covariance matrix Sigma

Available since 2.12

`real` **`multi_normal_lupdf`**`(row_vectors y | vectors mu, matrix Sigma)`
The log of the multivariate normal density of row vector(s) y given location vector(s) mu and covariance matrix Sigma dropping constant additive terms

Available since 2.25

`real` **`multi_normal_lpdf`**`(row_vectors y | row_vectors mu, matrix Sigma)`
The log of the multivariate normal density of row vector(s) y given location row vector(s) mu and covariance matrix Sigma

Available since 2.12

`real` **`multi_normal_lupdf`**`(row_vectors y | row_vectors mu, matrix Sigma)`
The log of the multivariate normal density of row vector(s) y given location row vector(s) mu and covariance matrix Sigma dropping constant additive terms

Available since 2.25

Although there is a direct multi-normal RNG function, if more than one result is required, it's much more efficient to Cholesky factor the covariance matrix and call `multi_normal_cholesky_rng`; see section multi-variate normal, cholesky parameterization.

`vector` **`multi_normal_rng`**`(vector mu, matrix Sigma)`
Generate a multivariate normal variate with location mu and covariance matrix Sigma; may only be used in transformed data and generated quantities blocks

Available since 2.0

`vector` **`multi_normal_rng`**`(row_vector mu, matrix Sigma)`
Generate a multivariate normal variate with location mu and covariance matrix Sigma; may only be used in transformed data and generated quantities blocks

Available since 2.18

`vectors` **`multi_normal_rng`**`(vectors mu, matrix Sigma)`
Generate an array of multivariate normal variates with locations mu and covariance matrix Sigma; may only be used in transformed data and generated quantities blocks

Available since 2.18

`vectors` **`multi_normal_rng`**`(row_vectors mu, matrix Sigma)`
Generate an array of multivariate normal variates with locations mu and covariance matrix Sigma; may only be used in transformed data and generated quantities blocks

Available since 2.18

25.2. Multivariate normal distribution, precision parameterization

Probability density function

If $K \in \mathbb{N}$, $\mu \in \mathbb{R}^K$, and $\Omega \in \mathbb{R}^{K \times K}$ is symmetric and positive definite, then for $y \in \mathbb{R}^K$,

$$\text{MultiNormalPrecision}(y|\mu, \Omega) = \text{MultiNormal}(y|\mu, \Omega^{-1})$$

Distribution statement

`y ~` **`multi_normal_prec`**`(mu, Omega)`

Increment target log probability density with `multi_normal_prec_lupdf(y | mu, Omega)`.

Available since 2.3

Stan functions

`real` **`multi_normal_prec_lpdf`**`(vectors y | vectors mu, matrix Omega)`
The log of the multivariate normal density of vector(s) y given location vector(s)

mu and positive definite precision matrix Omega

Available since 2.18

`real` **`multi_normal_prec_lupdf`**`(vectors y | vectors mu, matrix Omega)`
The log of the multivariate normal density of vector(s) y given location vector(s) mu and positive definite precision matrix Omega dropping constant additive terms

Available since 2.25

`real` **`multi_normal_prec_lpdf`**`(vectors y | row_vectors mu, matrix Omega)`
The log of the multivariate normal density of vector(s) y given location row vector(s) mu and positive definite precision matrix Omega

Available since 2.18

`real` **`multi_normal_prec_lupdf`**`(vectors y | row_vectors mu, matrix Omega)`
The log of the multivariate normal density of vector(s) y given location row vector(s) mu and positive definite precision matrix Omega dropping constant additive terms

Available since 2.25

`real` **`multi_normal_prec_lpdf`**`(row_vectors y | vectors mu, matrix Omega)`
The log of the multivariate normal density of row vector(s) y given location vector(s) mu and positive definite precision matrix Omega

Available since 2.18

`real` **`multi_normal_prec_lupdf`**`(row_vectors y | vectors mu, matrix Omega)`
The log of the multivariate normal density of row vector(s) y given location vector(s) mu and positive definite precision matrix Omega dropping constant additive terms

Available since 2.25

`real` **`multi_normal_prec_lpdf`**`(row_vectors y | row_vectors mu, matrix Omega)`
The log of the multivariate normal density of row vector(s) y given location row vector(s) mu and positive definite precision matrix Omega

Available since 2.18

real **multi_normal_prec_lupdf**(row_vectors y | row_vectors mu, matrix Omega)
The log of the multivariate normal density of row vector(s) y given location row vector(s) mu and positive definite precision matrix Omega dropping constant additive terms

Available since 2.25

25.3. Multivariate normal distribution, Cholesky parameterization

Probability density function

If $K \in \mathbb{N}$, $\mu \in \mathbb{R}^K$, and $L \in \mathbb{R}^{K \times K}$ is lower triangular and such that $LL^\top$ is positive definite, then for $y \in \mathbb{R}^K$,

$$\text{MultiNormalCholesky}(y|\mu, L) = \text{MultiNormal}(y|\mu, LL^\top).$$

If L is lower triangular and LL^{top} is a $K \times K$ positive definite matrix, then $L_{k,k}$ must be strictly positive for $k \in 1{:}K$. If an L is provided that is not the Cholesky factor of a positive-definite matrix, the probability functions will raise errors.

Distribution statement

y ~ **multi_normal_cholesky**(mu, L)

Increment target log probability density with `multi_normal_cholesky_lupdf(y | mu, L)`.

Available since 2.0

Stan functions

real **multi_normal_cholesky_lpdf**(vectors y | vectors mu, matrix L)
The log of the multivariate normal density of vector(s) y given location vector(s) mu and lower-triangular Cholesky factor of the covariance matrix L

Available since 2.18

real **multi_normal_cholesky_lupdf**(vectors y | vectors mu, matrix L)
The log of the multivariate normal density of vector(s) y given location vector(s) mu and lower-triangular Cholesky factor of the covariance matrix L dropping constant additive terms

Available since 2.25

real **multi_normal_cholesky_lpdf**(vectors y | row_vectors mu, matrix L)

The log of the multivariate normal density of vector(s) y given location row vector(s) mu and lower-triangular Cholesky factor of the covariance matrix L

Available since 2.18

`real` **`multi_normal_cholesky_lupdf`**`(vectors y | row_vectors mu, matrix L)`
The log of the multivariate normal density of vector(s) y given location row vector(s) mu and lower-triangular Cholesky factor of the covariance matrix L dropping constant additive terms

Available since 2.25

`real` **`multi_normal_cholesky_lpdf`**`(row_vectors y | vectors mu, matrix L)`
The log of the multivariate normal density of row vector(s) y given location vector(s) mu and lower-triangular Cholesky factor of the covariance matrix L

Available since 2.18

`real` **`multi_normal_cholesky_lupdf`**`(row_vectors y | vectors mu, matrix L)`
The log of the multivariate normal density of row vector(s) y given location vector(s) mu and lower-triangular Cholesky factor of the covariance matrix L dropping constant additive terms

Available since 2.25

`real` **`multi_normal_cholesky_lpdf`**`(row_vectors y | row_vectors mu, matrix L)`
The log of the multivariate normal density of row vector(s) y given location row vector(s) mu and lower-triangular Cholesky factor of the covariance matrix L

Available since 2.18

`real` **`multi_normal_cholesky_lupdf`**`(row_vectors y | row_vectors mu, matrix L)`
The log of the multivariate normal density of row vector(s) y given location row vector(s) mu and lower-triangular Cholesky factor of the covariance matrix L dropping constant additive terms

Available since 2.25

`vector` **`multi_normal_cholesky_rng`**`(vector mu, matrix L)`
Generate a multivariate normal variate with location mu and lower-triangular

Cholesky factor of the covariance matrix L; may only be used in transformed data and generated quantities blocks

Available since 2.3

`vector` **`multi_normal_cholesky_rng`**`(row_vector mu, matrix L)`
Generate a multivariate normal variate with location mu and lower-triangular Cholesky factor of the covariance matrix L; may only be used in transformed data and generated quantities blocks

Available since 2.18

`vectors` **`multi_normal_cholesky_rng`**`(vectors mu, matrix L)`
Generate an array of multivariate normal variates with locations mu and lower-triangular Cholesky factor of the covariance matrix L; may only be used in transformed data and generated quantities blocks

Available since 2.18

`vectors` **`multi_normal_cholesky_rng`**`(row_vectors mu, matrix L)`
Generate an array of multivariate normal variates with locations mu and lower-triangular Cholesky factor of the covariance matrix L; may only be used in transformed data and generated quantities blocks

Available since 2.18

25.4. Multivariate Gaussian process distribution

Probability density function

If $K, N \in \mathbb{N}$, $\Sigma \in \mathbb{R}^{N \times N}$ is symmetric, positive definite kernel matrix and $w \in \mathbb{R}^{K}$ is a vector of positive inverse scales, then for $y \in \mathbb{R}^{K \times N}$,

$$\text{MultiGP}(y|\Sigma, w) = \prod_{i=1}^{K} \text{MultiNormal}(y_i|0, w_i^{-1}\Sigma),$$

where y_i is the ith row of y. This is used to efficiently handle Gaussian Processes with multi-variate outputs where only the output dimensions share a kernel function but vary based on their scale. Note that this function does not take into account the mean prediction.

Distribution statement

`y ~` **`multi_gp`**`(Sigma, w)`

Increment target log probability density with `multi_gp_lupdf(y | Sigma, w)`.

Available since 2.3

Stan functions

`real` **`multi_gp_lpdf`**`(matrix y | matrix Sigma, vector w)`
The log of the multivariate GP density of matrix y given kernel matrix Sigma and inverses scales w

Available since 2.12

`real` **`multi_gp_lupdf`**`(matrix y | matrix Sigma, vector w)`
The log of the multivariate GP density of matrix y given kernel matrix Sigma and inverses scales w dropping constant additive terms

Available since 2.25

25.5. Multivariate Gaussian process distribution, Cholesky parameterization

Probability density function

If $K, N \in \mathbb{N}$, $L \in \mathbb{R}^{N\times N}$ is lower triangular and such that $LL^\top$ is positive definite kernel matrix (implying $L_{n,n} > 0$ for $n \in 1{:}N$), and $w \in \mathbb{R}^K$ is a vector of positive inverse scales, then for $y \in \mathbb{R}^{K\times N}$,

$$\text{MultiGPCholesky}(y \mid L, w) = \prod_{i=1}^{K} \text{MultiNormal}(y_i|0, w_i^{-1} LL^\top),$$

where y_i is the ith row of y. This is used to efficiently handle Gaussian Processes with multi-variate outputs where only the output dimensions share a kernel function but vary based on their scale. If the model allows parameterization in terms of Cholesky factor of the kernel matrix, this distribution is also more efficient than MultiGP(). Note that this function does not take into account the mean prediction.

Distribution statement

`y ~` **`multi_gp_cholesky`**`(L, w)`

Increment target log probability density with `multi_gp_cholesky_lupdf(y | L, w)`.

Available since 2.5

Stan functions

`real` **`multi_gp_cholesky_lpdf`**`(matrix y | matrix L, vector w)`
The log of the multivariate GP density of matrix y given lower-triangular Cholesky factor of the kernel matrix L and inverses scales w

Available since 2.12

`real` **`multi_gp_cholesky_lupdf`**`(matrix y | matrix L, vector w)`
The log of the multivariate GP density of matrix y given lower-triangular Cholesky factor of the kernel matrix L and inverses scales w dropping constant additive terms

Available since 2.25

25.6. Multivariate Student-t distribution

Probability density function

If $K \in \mathbb{N}$, $\nu \in \mathbb{R}^+$, $\mu \in \mathbb{R}^K$, and $\Sigma \in \mathbb{R}^{K \times K}$ is symmetric and positive definite, then for $y \in \mathbb{R}^K$,

$$\begin{aligned}&\text{MultiStudentT}(y \mid \nu, \mu, \Sigma)\\&= \frac{1}{\pi^{K/2}} \, \frac{1}{\nu^{K/2}} \, \frac{\Gamma((\nu+K)/2)}{\Gamma(\nu/2)} \, \frac{1}{\sqrt{|\Sigma|}} \, \left(1 + \frac{1}{\nu} \, (y - \mu)^\top \, \Sigma^{-1} \, (y - \mu)\right)^{-(\nu+K)/2}.\end{aligned}$$

Distribution statement

`y ~` **`multi_student_t`**`(nu, mu, Sigma)`

Increment target log probability density with `multi_student_t_lupdf(y | nu, mu, Sigma)`.

Available since 2.0

Stan functions

`real` **`multi_student_t_lpdf`**`(vectors y | real nu, vectors mu, matrix Sigma)`
The log of the multivariate Student-*t* density of vector(s) y given degrees of freedom nu, location vector(s) mu, and scale matrix Sigma

Available since 2.18

`real` **`multi_student_t_lupdf`**`(vectors y | real nu, vectors mu, matrix Sigma)`
The log of the multivariate Student-*t* density of vector(s) y given degrees of freedom nu, location vector(s) mu, and scale matrix Sigma dropping constant additive terms

Available since 2.25

`real` **`multi_student_t_lpdf`**`(vectors y | real nu, row_vectors mu, matrix Sigma)`

The log of the multivariate Student-*t* density of vector(s) y given degrees of freedom nu, location row vector(s) mu, and scale matrix Sigma

Available since 2.18

`real` **`multi_student_t_lupdf`**`(vectors y | real nu, row_vectors mu, matrix Sigma)`
The log of the multivariate Student-*t* density of vector(s) y given degrees of freedom nu, location row vector(s) mu, and scale matrix Sigma dropping constant additive terms

Available since 2.25

`real` **`multi_student_t_lpdf`**`(row_vectors y | real nu, vectors mu, matrix Sigma)`
The log of the multivariate Student-*t* density of row vector(s) y given degrees of freedom nu, location vector(s) mu, and scale matrix Sigma

Available since 2.18

`real` **`multi_student_t_lupdf`**`(row_vectors y | real nu, vectors mu, matrix Sigma)`
The log of the multivariate Student-*t* density of row vector(s) y given degrees of freedom nu, location vector(s) mu, and scale matrix Sigma dropping constant additive terms

Available since 2.25

`real` **`multi_student_t_lpdf`**`(row_vectors y | real nu, row_vectors mu, matrix Sigma)`
The log of the multivariate Student-*t* density of row vector(s) y given degrees of freedom nu, location row vector(s) mu, and scale matrix Sigma

Available since 2.18

`real` **`multi_student_t_lupdf`**`(row_vectors y | real nu, row_vectors mu, matrix Sigma)`
The log of the multivariate Student-*t* density of row vector(s) y given degrees of freedom nu, location row vector(s) mu, and scale matrix Sigma dropping constant additive terms

Available since 2.25

`vector` **`multi_student_t_rng`**`(real nu, vector mu, matrix Sigma)`
Generate a multivariate Student-*t* variate with degrees of freedom nu, location

mu, and scale matrix Sigma; may only be used in transformed data and generated quantities blocks

Available since 2.0

`vector` **`multi_student_t_rng`**`(real nu, row_vector mu, matrix Sigma)`
Generate a multivariate Student-t variate with degrees of freedom nu, location mu, and scale matrix Sigma; may only be used in transformed data and generated quantities blocks

Available since 2.18

`vectors` **`multi_student_t_rng`**`(real nu, vectors mu, matrix Sigma)`
Generate an array of multivariate Student-t variates with degrees of freedom nu, locations mu, and scale matrix Sigma; may only be used in transformed data and generated quantities blocks

Available since 2.18

`vectors` **`multi_student_t_rng`**`(real nu, row_vectors mu, matrix Sigma)`
Generate an array of multivariate Student-t variates with degrees of freedom nu, locations mu, and scale matrix Sigma; may only be used in transformed data andgenerated quantities blocks

Available since 2.18

25.7. Multivariate Student-t distribution, Cholesky parameterization

Probability density function

Let $K \in \mathbb{N}$, $\nu \in \mathbb{R}^+$, $\mu \in \mathbb{R}^K$, and L a $K \times K$ lower-triangular matrix with strictly positive, finite diagonal then

$$\begin{aligned}&\text{MultiStudentTCholesky}(y \mid \nu, \mu, L) \\ &= \frac{1}{\pi^{K/2}} \, \frac{1}{\nu^{K/2}} \, \frac{\Gamma((\nu+K)/2)}{\Gamma(\nu/2)} \, \frac{1}{|L|} \, \left(1 + \frac{1}{\nu} \, (y-\mu)^\top \, L^{-T} L^{-1} \, (y-\mu)\right)^{-(\nu+K)/2}.\end{aligned}$$

Distribution statement

`y ~` **`multi_student_t_cholesky`**`(nu, mu, L)`

Increment target log probability density with `multi_student_t_cholesky_lupdf(y | nu, mu, L)`.

Available since 2.30

Stan functions

`real` **`multi_student_t_cholesky_lpdf`**`(vectors y | real nu, vectors mu, matrix L)`
The log of the multivariate Student-t density of vector or array of vectors `y` given degrees of freedom `nu`, location vector or array of vectors `mu`, and Cholesky factor of the scale matrix `L`. For a definition of the arguments compatible with the `vectors` type, see the probability vectorization section.

Available since 2.30

`real` **`multi_student_t_cholesky_lupdf`**`(vectors y | real nu, vectors mu, matrix L)`
The log of the multivariate Student-t density of vector or vector array `y` given degrees of freedom `nu`, location vector or vector array `mu`, and Cholesky factor of the scale matrix `L`, dropping constant additive terms. For a definition of arguments compatible with the `vectors` type, see the probability vectorization section.

Available since 2.30

`vector` **`multi_student_t_cholesky_rng`**`(real nu, vector mu, matrix L)`
Generate a multivariate Student-t variate with degrees of freedom `nu`, location `mu`, and Cholesky factor of the scale matrix `L`; may only be used in transformed data and generated quantities blocks.

Available since 2.30

`array[] vector` **`multi_student_t_cholesky_rng`**`(real nu, array[] vector mu, matrix L)`
Generate a multivariate Student-t variate with degrees of freedom `nu`, location array `mu`, and Cholesky factor of the scale matrix `L`; may only be used in transformed data and generated quantities blocks.

Available since 2.30

`array[] vector` **`multi_student_t_cholesky_rng`**`(real nu, array[] row_vector mu, matrix L)`
Generate an array of multivariate Student-t variate with degrees of freedom `nu`, location array `mu`, and Cholesky factor of the scale matrix `L`; may only be used in transformed data and generated quantities blocks.

Available since 2.30

25.8. Gaussian dynamic linear models

A Gaussian Dynamic Linear model is defined as follows, For $t \in 1, \ldots, T$,

$$\begin{aligned} y_t &\sim N(F'\theta_t, V) \\ \theta_t &\sim N(G\theta_{t-1}, W) \\ \theta_0 &\sim N(m_0, C_0) \end{aligned}$$

where y is $n \times T$ matrix where rows are variables and columns are observations. These functions calculate the log-density of the observations marginalizing over the latent states ($p(y|F, G, V, W, m_0, C_0)$). This log-density is a system that is calculated using the Kalman Filter. If V is diagonal, then a more efficient algorithm which sequentially processes observations and avoids a matrix inversions can be used (Durbin and Koopman 2001, sec. 6.4).

Distribution statement

`y ~ `**`gaussian_dlm_obs`**`(F, G, V, W, m0, C0)`

Increment target log probability density with `gaussian_dlm_obs_lupdf(y | F, G, V, W, m0, C0)`.

Available since 2.0

Stan functions

The following two functions differ in the type of their V, the first taking a full observation covariance matrix V and the second a vector V representing the diagonal of the observation covariance matrix. The sampling statement defined in the previous section works with either type of observation V.

`real` **`gaussian_dlm_obs_lpdf`**`(matrix y | matrix F, matrix G, matrix V, matrix W, vector m0, matrix C0)`
The log of the density of the Gaussian Dynamic Linear model with observation matrix y in which rows are variables and columns are observations, design matrix F, transition matrix G, observation covariance matrix V, system covariance matrix W, and the initial state is distributed normal with mean m0 and covariance C0.

Available since 2.12

`real` **`gaussian_dlm_obs_lupdf`**`(matrix y | matrix F, matrix G, matrix V, matrix W, vector m0, matrix C0)`
The log of the density of the Gaussian Dynamic Linear model with observation matrix y in which rows are variables and columns are observations, design matrix F, transition matrix G, observation covariance matrix V, system covariance matrix

W, and the initial state is distributed normal with mean m0 and covariance C0. This function drops constant additive terms.

Available since 2.25

`real` **`gaussian_dlm_obs_lpdf`**`(matrix y | matrix F, matrix G, vector V, matrix W, vector m0, matrix C0)`
The log of the density of the Gaussian Dynamic Linear model with observation matrix y in which rows are variables and columns are observations, design matrix F, transition matrix G, observation covariance matrix with diagonal V, system covariance matrix W, and the initial state is distributed normal with mean m0 and covariance C0.

Available since 2.12

`real` **`gaussian_dlm_obs_lupdf`**`(matrix y | matrix F, matrix G, vector V, matrix W, vector m0, matrix C0)`
The log of the density of the Gaussian Dynamic Linear model with observation matrix y in which rows are variables and columns are observations, design matrix F, transition matrix G, observation covariance matrix with diagonal V, system covariance matrix W, and the initial state is distributed normal with mean m0 and covariance C0. This function drops constant additive terms.

Available since 2.25

26. Simplex Distributions

The simplex probabilities have support on the unit K-simplex for a specified K. A K-dimensional vector θ is a unit K-simplex if $\theta_k \geq 0$ for $k \in \{1, \ldots, K\}$ and $\sum_{k=1}^{K} \theta_k = 1$.

26.1. Dirichlet distribution

Probability density function

If $K \in \mathbb{N}$ and $\alpha \in (\mathbb{R}^+)^K$, then for $\theta \in K$-simplex,

$$\text{Dirichlet}(\theta|\alpha) = \frac{\Gamma\left(\sum_{k=1}^{K} \alpha_k\right)}{\prod_{k=1}^{K} \Gamma(\alpha_k)} \prod_{k=1}^{K} \theta_k^{\alpha_k - 1}$$

Warning: If any of the components of θ satisfies $\theta_i = 0$ or $\theta_i = 1$, then the probability is 0 and the log probability is $-\infty$. Similarly, the distribution requires strictly positive parameters, with $\alpha_i > 0$ for each i.

Meaning of Dirichlet parameters

A symmetric Dirichlet prior is $[\alpha, \ldots, \alpha]^\top$. To code this in Stan,

```
data {
  int<lower=1> K;
  real<lower=0> alpha;
}
generated quantities {
  vector[K] theta = dirichlet_rng(rep_vector(alpha, K));
}
```

Taking $K = 10$, here are the first five draws for $\alpha = 1$. For $\alpha = 1$, the distribution is uniform over simplexes.

```
1) 0.17 0.05 0.07 0.17 0.03 0.13 0.03 0.03 0.27 0.05
2) 0.08 0.02 0.12 0.07 0.52 0.01 0.07 0.04 0.01 0.06
3) 0.02 0.03 0.22 0.29 0.17 0.10 0.09 0.00 0.05 0.03
4) 0.04 0.03 0.21 0.13 0.04 0.01 0.10 0.04 0.22 0.18
5) 0.11 0.22 0.02 0.01 0.06 0.18 0.33 0.04 0.01 0.01
```

That does not mean it's uniform over the marginal probabilities of each element. As the size of the simplex grows, the marginal draws become more and more concentrated below (not around) $1/K$. When one component of the simplex is large, the others must all be relatively small to compensate. For example, in a uniform distribution on 10-simplexes, the probability that a component is greater than the mean of $1/10$ is only 39%. Most of the posterior marginal probability mass for each component is in the interval $(0, 0.1)$.

When the α value is small, the draws gravitate to the corners of the simplex. Here are the first five draws for $\alpha = 0.001$.

```
1) 3e-203 0e+00 2e-298 9e-106 1e+000 0e+00 0e+000 1e-047 0e+00 4e-279
2) 1e+000 0e+00 5e-279 2e-014 1e-275 0e+00 3e-285 9e-147 0e+00 0e+000
3) 1e-308 0e+00 1e-213 0e+000 0e+000 8e-75 0e+000 1e+000 4e-58 7e-112
4) 6e-166 5e-65 3e-068 3e-147 0e+000 1e+00 3e-249 0e+000 0e+00 0e+000
5) 2e-091 0e+00 0e+000 0e+000 1e-060 0e+00 4e-312 1e+000 0e+00 0e+000
```

Each row denotes a draw. Each draw has a single value that rounds to one and other values that are very close to zero or rounded down to zero.

As α increases, the draws become increasingly uniform. For $\alpha = 1000$,

```
1) 0.10 0.10 0.10 0.10 0.10 0.10 0.10 0.10 0.10 0.10
2) 0.10 0.10 0.09 0.10 0.10 0.10 0.11 0.10 0.10 0.10
3) 0.10 0.10 0.10 0.10 0.10 0.10 0.10 0.10 0.10 0.10
4) 0.10 0.10 0.10 0.10 0.10 0.10 0.10 0.10 0.10 0.10
5) 0.10 0.10 0.10 0.10 0.10 0.10 0.10 0.10 0.10 0.10
```

Distribution statement

`theta ~ dirichlet(alpha)`

Increment target log probability density with `dirichlet_lupdf(theta | alpha)`.

Available since 2.0

Stan functions

The Dirichlet probability functions are overloaded to allow the simplex θ and prior counts (plus one) α to be vectors or row vectors (or to mix the two types). The density functions are also vectorized, so they allow arrays of row vectors or vectors as arguments; see section vectorized function signatures for a description of vectorization.

`real` **`dirichlet_lpdf`**`(vectors theta | vectors alpha)`
The log of the Dirichlet density for simplex(es) `theta` given prior counts (plus one) `alpha`

Available since 2.12, vectorized in 2.21

`real` **`dirichlet_lupdf`**`(vectors theta | vectors alpha)`
The log of the Dirichlet density for simplex(es) `theta` given prior counts (plus one) `alpha` dropping constant additive terms

Available since 2.25

`vector` **`dirichlet_rng`**`(vector alpha)`
Generate a Dirichlet variate with prior counts (plus one) `alpha`; may only be used in transformed data and generated quantities blocks

Available since 2.0

27. Correlation Matrix Distributions

The correlation matrix distributions have support on the (Cholesky factors of) correlation matrices. A Cholesky factor L for a $K \times K$ correlation matrix Σ of dimension K has rows of unit length so that the diagonal of $LL^\top$ is the unit K-vector. Even though models are usually conceptualized in terms of correlation matrices, it is better to operationalize them in terms of their Cholesky factors. If you are interested in the posterior distribution of the correlations, you can recover them in the generated quantities block via

```
generated quantities {
  corr_matrix[K] Sigma;
  Sigma = multiply_lower_tri_self_transpose(L);
}
```

27.1. LKJ correlation distribution

Probability density function

For $\eta > 0$, if Σ a positive-definite, symmetric matrix with unit diagonal (i.e., a correlation matrix), then

$$\text{LkjCorr}(\Sigma|\eta) \propto \det(\Sigma)^{(\eta-1)}.$$

The expectation is the identity matrix for any positive value of the shape parameter η, which can be interpreted like the shape parameter of a symmetric beta distribution:

- if $\eta = 1$, then the density is uniform over correlation matrices of order K;
- if $\eta > 1$, the identity matrix is the modal correlation matrix, with a sharper peak in the density at the identity matrix for larger η; and
- for $0 < \eta < 1$, the density has a trough at the identity matrix.
- if η were an unknown parameter, the Jeffreys prior is proportional to $\sqrt{2\sum_{k=1}^{K-1}\left(\psi_1\left(\eta + \frac{K-k-1}{2}\right) - 2\psi_1\left(2\eta + K - k - 1\right)\right)}$, where $\psi_1()$ is the trigamma function

See (Lewandowski, Kurowicka, and Joe 2009) for definitions. However, it is much better computationally to work directly with the Cholesky factor of Σ, so this distribution should never be explicitly used in practice.

Distribution statement

`y ~ lkj_corr(eta)`

Increment target log probability density with `lkj_corr_lupdf(y | eta)`.

Available since 2.3

Stan functions

`real` **`lkj_corr_lpdf`**`(matrix y | real eta)`
The log of the LKJ density for the correlation matrix y given nonnegative shape eta. `lkj_corr_cholesky_lpdf` is faster, more numerically stable, uses less memory, and should be preferred to this.

Available since 2.12

`real` **`lkj_corr_lupdf`**`(matrix y | real eta)`
The log of the LKJ density for the correlation matrix y given nonnegative shape eta dropping constant additive terms. `lkj_corr_cholesky_lupdf` is faster, more numerically stable, uses less memory, and should be preferred to this.

Available since 2.25

`matrix` **`lkj_corr_rng`**`(int K, real eta)`
Generate a LKJ random correlation matrix of order K with shape eta; may only be used in transformed data and generated quantities blocks

Available since 2.0

27.2. Cholesky LKJ correlation distribution

Stan provides an implicit parameterization of the LKJ correlation matrix density in terms of its Cholesky factor, which you should use rather than the explicit parameterization in the previous section. For example, if `L` is a Cholesky factor of a correlation matrix, then

```
L ~ lkj_corr_cholesky(2.0); # implies L * L' ~ lkj_corr(2.0);
```

Because Stan requires models to have support on all valid constrained parameters, `L` will almost always[1] be a parameter declared with the type of a Cholesky factor for a correlation matrix; for example,

[1]It is possible to build up a valid `L` within Stan, but that would then require Jacobian adjustments to imply the intended posterior.

```
parameters {   cholesky_factor_corr[K] L;   # rather than corr_matrix[K]
```

Probability density function

For $\eta > 0$, if L is a $K \times K$ lower-triangular Cholesky factor of a symmetric positive-definite matrix with unit diagonal (i.e., a correlation matrix), then

$$\text{LkjCholesky}(L|\eta) \propto |J| \det(LL^\top)^{(\eta-1)} = \prod_{k=2}^{K} L_{kk}^{K-k+2\eta-2}.$$

See the previous section for details on interpreting the shape parameter η. Note that even if $\eta = 1$, it is still essential to evaluate the density function because the density of L is not constant, regardless of the value of η, even though the density of $LL^\top$ is constant iff $\eta = 1$.

A lower triangular L is a Cholesky factor for a correlation matrix if and only if $L_{k,k} > 0$ for $k \in 1{:}K$ and each row L_k has unit Euclidean length.

Distribution statement

`L ~ `**`lkj_corr_cholesky`**`(eta)`

Increment target log probability density with `lkj_corr_cholesky_lupdf(L | eta)`.

Available since 2.4

Stan functions

`real` **`lkj_corr_cholesky_lpdf`**`(matrix L | real eta)`
The log of the LKJ density for the lower-triangular Cholesky factor L of a correlation matrix given shape eta

Available since 2.12

`real` **`lkj_corr_cholesky_lupdf`**`(matrix L | real eta)`
The log of the LKJ density for the lower-triangular Cholesky factor L of a correlation matrix given shape eta dropping constant additive terms

Available since 2.25

`matrix` **`lkj_corr_cholesky_rng`**`(int K, real eta)`
Generate a random Cholesky factor of a correlation matrix of order K that is distributed LKJ with shape eta; may only be used in transformed data and generated quantities blocks

Available since 2.4

28. Covariance Matrix Distributions

The covariance matrix distributions have support on symmetric, positive-definite $K \times K$ matrices or their Cholesky factors (square, lower triangular matrices with positive diagonal elements).

28.1. Wishart distribution

Probability density function

If $K \in \mathbb{N}$, $\nu \in (K-1, \infty)$, and $S \in \mathbb{R}^{K \times K}$ is symmetric and positive definite, then for symmetric and positive-definite $W \in \mathbb{R}^{K \times K}$,

$$\text{Wishart}(W \mid \nu, S) = \frac{1}{2^{\nu K/2}} \frac{1}{\Gamma_K\left(\frac{\nu}{2}\right)} \; |S|^{-\nu/2} \; |W|^{(\nu-K-1)/2} \; \exp\left(-\frac{1}{2}\,\text{tr}\left(S^{-1}W\right)\right),$$

where tr() is the matrix trace function, and $\Gamma_K()$ is the multivariate Gamma function,

$$\Gamma_K(x) = \frac{1}{\pi^{K(K-1)/4}} \prod_{k=1}^{K} \Gamma\left(x + \frac{1-k}{2}\right).$$

Distribution statement

```
W ~ wishart(nu, Sigma)
```

Increment target log probability density with `wishart_lupdf(W | nu, Sigma)`.

Available since 2.0

Stan functions

`real` **`wishart_lpdf`**`(matrix W | real nu, matrix Sigma)`
Return the log of the Wishart density for symmetric and positive-definite matrix `W` given degrees of freedom `nu` and symmetric and positive-definite scale matrix `Sigma`.

Available since 2.12

`real` **`wishart_lupdf`**`(matrix W | real nu, matrix Sigma)`
Return the log of the Wishart density for symmetric and positive-definite matrix `W` given degrees of freedom `nu` and symmetric and positive-definite scale matrix `Sigma` dropping constant additive terms.

Available since 2.25

`matrix` **`wishart_rng`**`(real nu, matrix Sigma)`
Generate a Wishart variate with degrees of freedom `nu` and symmetric and positive-definite scale matrix `Sigma`; may only be used in transformed data and generated quantities blocks.

Available since 2.0

28.2. Wishart distribution, Cholesky Parameterization

The Cholesky parameterization of the Wishart distribution uses a Cholesky factor for both the variate and the parameter. If S and W are positive definite matrices with Cholesky factors L_S and L_W (i.e., $S = L_S L_S^\top$ and $W = L_W L_W^\top$), then the Cholesky parameterization is defined so that

$$L_W \sim \text{WishartCholesky}(\nu, L_S)$$

if and only if

$$W \sim \text{Wishart}(\nu, S).$$

Probability density function

If $K \in \mathbb{N}$, $\nu \in (K-1, \infty)$, and $L_S, L_W \in \mathbb{R}^{K \times K}$ are lower triangular matrixes with positive diagonal elements, then the Cholesky parameterized Wishart density is

$$\text{WishartCholesky}(L_W \mid \nu, L_S) = \text{Wishart}(L_W L_W^\top \mid \nu, L_S L_S^\top) \left| J_{f^{-1}} \right|,$$

where $J_{f^{-1}}$ is the Jacobian of the (inverse) transform of the variate, $f^{-1}(L_W) = L_W L_W^\top$. The log absolute determinant is

$$\log \left| J_{f^{-1}} \right| = K \log(2) + \sum_{k=1}^{K} (K - k + 1) \log{(L_W)_{k,k}}.$$

The probability functions will raise errors if $\nu \leq K - 1$ or if L_S and L_W are not Cholesky factors (square, lower-triangular matrices with positive diagonal elements) of the same size.

Stan functions

`real` **`wishart_cholesky_lpdf`**`(matrix L_W | real nu, matrix L_S)`
Return the log of the Wishart density for lower-triangular Cholesky factor `L_W` given degrees of freedom `nu` and lower-triangular Cholesky factor of the scale matrix `L_S`.

Available since 2.30

`real` **`wishart_cholesky_lupdf`**`(matrix L_W | real nu, matrix L_S)`
Return the log of the Wishart density for lower-triangular Cholesky factor of `L_W` given degrees of freedom `nu` and lower-triangular Cholesky factor of the scale matrix `L_S` dropping constant additive terms.

Available since 2.30

`matrix` **`wishart_cholesky_rng`**`(real nu, matrix L_S)`
Generate the Cholesky factor of a Wishart variate with degrees of freedom `nu` and lower-triangular Cholesky factor of the scale matrix `L_S`; may only be used in transformed data and generated quantities blocks

Available since 2.30

28.3. Inverse Wishart distribution

Probability density function

If $K \in \mathbb{N}$, $\nu \in (K-1, \infty)$, and $S \in \mathbb{R}^{K \times K}$ is symmetric and positive definite, then for symmetric and positive-definite $W \in \mathbb{R}^{K \times K}$,

$$\text{InvWishart}(W \mid \nu, S) = \frac{1}{2^{\nu K/2}} \, \frac{1}{\Gamma_K\left(\frac{\nu}{2}\right)} \, |S|^{\nu/2} \, |W|^{-(\nu+K+1)/2} \, \exp\left(-\frac{1}{2}\,\text{tr}(SW^{-1})\right).$$

Distribution statement

`W ~` **`inv_wishart`**`(nu, Sigma)`

Increment target log probability density with `inv_wishart_lupdf(W | nu, Sigma)`.

Available since 2.0

Stan functions

`real` **`inv_wishart_lpdf`**`(matrix W | real nu, matrix Sigma)`
Return the log of the inverse Wishart density for symmetric and positive-definite matrix `W` given degrees of freedom `nu` and symmetric and positive-definite scale matrix `Sigma`.

Available since 2.12

`real` **`inv_wishart_lupdf`**`(matrix W | real nu, matrix Sigma)`
Return the log of the inverse Wishart density for symmetric and positive-definite

matrix `W` given degrees of freedom `nu` and symmetric and positive-definite scale matrix `Sigma` dropping constant additive terms.

Available since 2.25

`matrix` **`inv_wishart_rng`**`(real nu, matrix Sigma)`
Generate an inverse Wishart variate with degrees of freedom `nu` and symmetric and positive-definite scale matrix `Sigma`; may only be used in transformed data and generated quantities blocks.

Available since 2.0

28.4. Inverse Wishart distribution, Cholesky Parameterization

The Cholesky parameterization of the inverse Wishart distribution uses a Cholesky factor for both the variate and the parameter. If S and W are positive definite matrices with Cholesky factors L_S and L_W (i.e., $S = L_S L_S^\top$ and $W = L_W L_W^\top$), then the Cholesky parameterization is defined so that

$$L_W \sim \text{InvWishartCholesky}(\nu, L_S)$$

if and only if

$$W \sim \text{InvWishart}(\nu, S).$$

Probability density function

If $K \in \mathbb{N}$, $\nu \in (K-1, \infty)$, and $L_S, L_W \in \mathbb{R}^{K \times K}$ are lower triangular matrixes with positive diagonal elements, then the Cholesky parameterized inverse Wishart density is

$$\text{InvWishartCholesky}(L_W \mid \nu, L_S) = \text{InvWishart}(L_W L_W^\top \mid \nu, L_S L_S^\top) \left| J_{f^{-1}} \right|,$$

where $J_{f^{-1}}$ is the Jacobian of the (inverse) transform of the variate, $f^{-1}(L_W) = L_W L_W^\top$. The log absolute determinant is

$$\log \left| J_{f^{-1}} \right| = K \log(2) + \sum_{k=1}^{K} (K - k + 1) \log{(L_W)_{k,k}}.$$

The probability functions will raise errors if $\nu \leq K - 1$ or if L_S and L_W are not Cholesky factors (square, lower-triangular matrices with positive diagonal elements) of the same size.

Stan functions

`real` **`inv_wishart_cholesky_lpdf`**`(matrix L_W | real nu, matrix L_S)`
Return the log of the inverse Wishart density for lower-triangular Cholesky factor `L_W` given degrees of freedom `nu` and lower-triangular Cholesky factor of the scale matrix `L_S`.

Available since 2.30

`real` **`inv_wishart_cholesky_lupdf`**`(matrix L_W | real nu, matrix L_S)`
Return the log of the inverse Wishart density for lower-triangular Cholesky factor of `L_W` given degrees of freedom `nu` and lower-triangular Cholesky factor of the scale matrix `L_S` dropping constant additive terms.

Available since 2.30

`matrix` **`inv_wishart_cholesky_rng`**`(real nu, matrix L_S)`
Generate the Cholesky factor of an inverse Wishart variate with degrees of freedom `nu` and lower-triangular Cholesky factor of the scale matrix `L_S`; may only be used in transformed data and generated quantities blocks.

Available since 2.30

Part IV

Additional Distributions

29. Hidden Markov Models

An elementary first-order Hidden Markov model is a probabilistic model over N observations, y_n, and N hidden states, x_n, which can be fully defined by the conditional distributions $p(y_n \mid x_n, \phi)$ and $p(x_n \mid x_{n-1}, \phi)$. Here we make the dependency on additional model parameters, ϕ, explicit. When x is continuous, the user can explicitly encode these distributions in Stan and use Markov chain Monte Carlo to integrate x out.

When each state x takes a value over a discrete and finite set, say $\{1, 2, ..., K\}$, we can take advantage of the dependency structure to marginalize x and compute $p(y \mid \phi)$. We start by defining the conditional observational distribution, stored in a $K \times N$ matrix ω with

$$\omega_{kn} = p(y_n \mid x_n = k, \phi).$$

Next, we introduce the $K \times K$ transition matrix, Γ, with

$$\Gamma_{ij} = p(x_n = j \mid x_{n-1} = i, \phi).$$

Each row defines a probability distribution and must therefore be a simplex (i.e. its components must add to 1). Currently, Stan only supports stationary transitions where a single transition matrix is used for all transitions. Finally we define the initial state K-vector ρ, with

$$\rho_k = p(x_0 = k \mid \phi).$$

The Stan functions that support this type of model are special in that the user does not explicitly pass y and ϕ as arguments. Instead, the user passes $\log \omega$, Γ, and ρ, which in turn depend on y and ϕ.

29.1. Stan functions

`real` **`hmm_marginal`**`(matrix log_omega, matrix Gamma, vector rho)`
Returns the log probability density of y, with x_n integrated out at each iteration.

Available since 2.24

The arguments represent (1) the log density of each output, (2) the transition matrix, and (3) the initial state vector.

- *`log_omega`*: $\log \omega_{kn} = \log p(y_n \mid x_n = k, \phi)$, log density of each output,

- `Gamma`: $\Gamma_{ij} = p(x_n = j | x_{n-1} = i, \phi)$, the transition matrix,
- `rho`: $\rho_k = p(x_0 = k \mid \phi)$, the initial state probability.

`array[] int` **`hmm_latent_rng`**`(matrix log_omega, matrix Gamma, vector rho)`
Returns a length N array of integers over $\{1, ..., K\}$, sampled from the joint posterior distribution of the hidden states, $p(x \mid \phi, y)$. May be only used in transformed data and generated quantities.

Available since 2.24

`matrix` **`hmm_hidden_state_prob`**`(matrix log_omega, matrix Gamma, vector rho)`
Returns the matrix of marginal posterior probabilities of each hidden state value. This will be a $K \times N$ matrix. The n^{th} column is a simplex of probabilities for the n^{th} variable. Moreover, let A be the output. Then $A_{ij} = p(x_j = i \mid \phi, y)$. This function may only be used in transformed data and generated quantities.

Available since 2.24

Part V

Appendix

30. Mathematical Functions

This appendix provides the definition of several mathematical functions used throughout the manual.

30.1. Beta

The beta function, $\mathrm{B}(a,b)$, computes the normalizing constant for the beta distribution, and is defined for $a > 0$ and $b > 0$ by

$$\mathrm{B}(a,b) \;=\; \int_0^1 u^{a-1}(1-u)^{b-1}\,du \;=\; \frac{\Gamma(a)\,\Gamma(b)}{\Gamma(a+b)}\,,$$

where $\Gamma(x)$ is the Gamma function.

30.2. Incomplete beta

The incomplete beta function, $\mathrm{B}(x;a,b)$, is defined for $x \in [0,1]$ and $a,b \geq 0$ such that $a+b \neq 0$ by

$$\mathrm{B}(x;\,a,b) \;=\; \int_0^x u^{a-1}\,(1-u)^{b-1}\,du,$$

where $\mathrm{B}(a,b)$ is the beta function defined in appendix. If $x = 1$, the incomplete beta function reduces to the beta function, $\mathrm{B}(1;a,b) = \mathrm{B}(a,b)$.

The regularized incomplete beta function divides the incomplete beta function by the beta function,

$$I_x(a,b) \;=\; \frac{\mathrm{B}(x;\,a,b)}{B(a,b)}\,.$$

30.3. Gamma

The gamma function, $\Gamma(x)$, is the generalization of the factorial function to continuous variables, defined so that for positive integers n,

$$\Gamma(n+1) = n!$$

Generalizing to all positive numbers and non-integer negative numbers,

$$\Gamma(x) = \int_0^\infty u^{x-1}\exp(-u)\,du.$$

30.4. Digamma

The digamma function Ψ is the derivative of the $\log \Gamma$ function,

$$\Psi(u) \;=\; \frac{d}{du}\log\Gamma(u) \;=\; \frac{1}{\Gamma(u)}\,\frac{d}{du}\Gamma(u).$$

References

Bailey, David H., Karthik Jeyabalan, and Xiaoye S. Li. 2005. "A Comparison of Three High-Precision Quadrature Schemes." *Experiment. Math.* 14 (3): 317–29. https://projecteuclid.org:443/euclid.em/1128371757.

Bowling, Shannon R., Mohammad T. Khasawneh, Sittichai Kaewkuekool, and Byung Rae Cho. 2009. "A Logistic Approximation to the Cumulative Normal Distribution." *Journal of Industrial Engineering and Management* 2 (1): 114–27.

Ding, Peng, and Joseph K. Blitzstein. 2018. "On the Gaussian Mixture Representation of the Laplace Distribution." *The American Statistician* 72 (2): 172–74. https://doi.org/10.1080/00031305.2017.1291448.

Durbin, J., and S. J. Koopman. 2001. *Time Series Analysis by State Space Methods*. New York: Oxford University Press.

Feller, William. 1968. *An Introduction to Probability Theory and Its Applications*. Vol. 1. 3. Wiley, New York.

Gaebler, Johann D. 2021. "Autodiff for Implicit Functions in Stan." https://www.jgaeb.com/2021/09/13/implicit-autodiff.html#fn:7.

Gelman, Andrew, J. B. Carlin, Hal S. Stern, David B. Dunson, Aki Vehtari, and Donald B. Rubin. 2013. *Bayesian Data Analysis*. Third Edition. London: Chapman & Hall / CRC Press.

Golub, G. H., and V. Pereyra. 1973. "The Differentiation of Pseudo-Inverses and Nonlinear Least Squares Problems Whose Variables Separate." *SIAM Journal on Numerical Analysis* 10 (2): 413–32. https://doi.org/10.1137/0710036.

Guennebaud, Gaël, Benoît Jacob, et al. 2010. "Eigen V3." http://eigen.tuxfamily.org.

Hindmarsh, Alan C, Peter N Brown, Keith E Grant, Steven L Lee, Radu Serban, Dan E Shumaker, and Carol S Woodward. 2005. "SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers." *ACM Transactions on Mathematical Software (TOMS)* 31 (3): 363–96.

Jorge J. More, Kenneth E. Hillstrom, Burton S. Garbow. 1980. *User Guide for MINPACK-1*. 9700 South Cass Avenue, Argonne, Illinois 60439: Argonne National Laboratory.

Lewandowski, Daniel, Dorota Kurowicka, and Harry Joe. 2009. "Generating Random Correlation Matrices Based on Vines and Extended Onion Method." *Journal of Multivariate Analysis* 100: 1989–2001.

Margossian, Charles C, and Michael Betancourt. 2022. "Efficient Automatic Differentiation of Implicit Functions." *Preprint. arXiv:2112.14217.*

Mori, Masatake. 1978. "An IMT-Type Double Exponential Formula for Numerical Integration." *Publications of the Research Institute for Mathematical Sciences* 14 (3): 713–29. https://doi.org/10.2977/prims/1195188835.

Navarro, Danielle J, and Ian G Fuss. 2009. "Fast and Accurate Calculations for First-Passage Times in Wiener Diffusion Models." *Journal of Mathematical Psychology* 53 (4): 222–30.

Powell, Michael J. D. 1970. "A Hybrid Method for Nonlinear Equations." In *Numerical Methods for Nonlinear Algebraic Equations*, edited by P. Rabinowitz. Gordon; Breach.

Serban, Radu, Cosmin Petra, Alan C. Hindmarsh, Cody J. Balos, David J. Gardner, Daniel R. Reynolds, and Carol S. Woodward. 2021. "User Documentation for IDAS V5.0.0." Lawrence Livermore National Laboratory.

Takahasi, Hidetosi, and Masatake Mori. 1974. "Double Exponential Formulas for Numerical Integration." *Publications of the Research Institute for Mathematical Sciences* 9 (3): 721–41. https://doi.org/10.2977/prims/1195192451.

Tanaka, Ken'ichiro, Masaaki Sugihara, Kazuo Murota, and Masatake Mori. 2009. "Function Classes for Double Exponential Integration Formulas." *Numerische Mathematik* 111 (4): 631–55. https://doi.org/10.1007/s00211-008-0195-1.

Vandekerckhove, Joachim, and Dominik Wabersich. 2014. "The RWiener Package: An R Package Providing Distribution Functions for the Wiener Diffusion Model." *The R Journal* 6/1. http://journal.r-project.org/archive/2014-1/vandekerckhove-wabersich.pdf.

Wichura, Michael J. 1988. "Algorithm AS 241: The Percentage Points of the Normal Distribution." *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 37 (3): 477–84. http://www.jstor.org/stable/2347330.

Index