mars Documentation

Release 0.2.1

jisheng,qinxing

Sep 23, 2019

Getting Started

Mars is a tensor-based unified framework for large-scale data computation.

CHAPTER 1

Mars tensor

documentation

Mars tensor provides a familiar interface like Numpy.

Numpy	Mars tensor

`import numpy as np`
`a = np.random.rand(1000, 2000)`
`(a + 1).sum(axis=1)`
	`import mars.tensor as mt`
	`a = mt.random.rand(1000, 2000)`
	`(a + 1).sum(axis=1).execute()`

CHAPTER 2

Easy to scale in and scale out

Mars can scale in to a single machine, and scale out to a cluster with hundreds of machines. Both the local and distributed version share the same piece of code, it's fairly simple to migrate from a single machine to a cluster due to the increase of data.

2.1 Standalone mode

2.1.1 Threaded

You can install Mars via pip:

```
pip install pymars
```

After installation, you can simply open a Python console and run

```
import mars.tensor as mt
from mars.session import new_session

a = mt.ones((5, 5), chunk_size=3)
b = a * 4
# if there isn't a local session,
# execute will create a default one first
b.execute()

# or create a session explicitly
sess = new_session()
sess.run(b)  # run b
```

2.1.2 Local cluster

Users can start the distributed runtime of Mars on a single machine. First, install Mars distributed by run

```
pip install 'pymars[distributed]'
```

For now, local cluster mode can only run on Linux and Mac OS.

Then start a local cluster by run

```
import mars.tensor as mt
from mars.deploy.local import new_cluster
from mars.session import new_session

cluster = new_cluster()

# new cluster will start a session and set it as default one
# execute will then run in the local cluster
a = mt.random.rand(10, 10)
a.dot(a.T).execute()

# cluster.session is the session created
cluster.session.run(a + 1)

# users can also create a session explicitly
# cluster.endpoint needs to be passed to new_session
session2 = new_session(cluster.endpoint)
session2.run(a * 2)
```

2.2 Run on Clusters

2.2.1 Basic Steps

Mars can be deployed on a cluster. First, you need to run

```
pip install 'pymars[distributed]'
```

on every node in the cluster. This will install dependencies needed for distributed execution on your cluster. After that, you may select a node as scheduler and another as web service, leaving other nodes as workers. The scheduler can be started with the following command:

```
mars-scheduler -a <scheduler_ip> -p <scheduler_port>
```

Web service can be started with the following command:

```
mars-web -a <web_ip> -p <web_port> -s <scheduler_ip>:<scheduler_port>
```

Workers can be started with the following command:

```
mars-worker -a <worker_ip> -p <worker_port> -s <scheduler_ip>:<scheduler_port>
```

After all Mars processes are started, you can open a Python console and run

```
import mars.tensor as mt
from mars.session import new_session
sess = new_session('http://<web_ip>:<web_port>')
a = mt.ones((2000, 2000), chunk_size=200)
b = mt.inner(a, a)
sess.run(b)
```

You can open a web browser and type `http://<web_ip>:<web_port>` to open Mars UI to look up resource usage of workers and execution progress of the task submitted just now.

2.2.2 Using Command Lines

When running Mars with command line, you can specify arguments to control the behavior of Mars processes. All Mars services have common arguments listed below.

Argument	Description
`-a`	Advertise address exposed to other processes in the cluster, useful when the server has multiple IP addresses, or the service is deployed inside a VM or container
`-H`	Service IP binding, `0.0.0.0` by default
`-p`	Port of the service. If absent, a randomized port will be used
`-s`	List of scheduler endpoints, separated by commas. Useful for workers and webs to spot schedulers, or when you want to run more than one schedulers
`--log-level`	Log level, can be `debug`, `info`, `warning`, `error`
`--log-format`	Log format, can be Python logging format
`--log-conf`	Python logging configuration file, `logging.conf` by default

Extra arguments for schedulers are listed below.

Argument	Description
`--nproc`	Number of processes. If absent, the value will be the available number of cores

Extra arguments for workers are listed below. Details about memory tuning can be found at the next section.

Argument	Description
`--cpu-procs`	Number of computation processes on CPUs. If absent, the value will be the available number of cores
`--net-procs`	Number of processes for network transfer. 4 by default
`--phy-mem`	Limit of physical memory, can be percentages of total memory or multiple of bytes. For instance, `4g` or `80%` are both acceptable. If absent, the size of physical memory will be used
`--cache-mem`	Size of shared memory, can be percentages of total memory or multiple of bytes. For instance, `4g` or `80%` are both acceptable. If absent, 50% of free memory will be used
`--min-mem`	Minimal free memory to start worker, can be percentages of total memory or multiple of bytes. For instance, `4g` or `80%` are both acceptable. `128m` by default
`--spill-dirs`	Directories to spill to, separated by : in MacOS or Linux.
`--plasma-dir`	Directory of plasma store. When specified, the size of plasma store will not be considered in memory management.

For instance, if you want to start a Mars cluster with two schedulers, two workers and one web service, you can run commands below (memory and CPU tunings are omitted):

On Scheduler 1 (192.168.1.10):

```
mars-scheduler -a 192.168.1.10 -p 7001 -s 192.168.1.10:7001,192.168.1.11:7002
```

On Scheduler 2 (192.168.1.11):

```
mars-scheduler -a 192.168.1.11 -p 7002 -s 192.168.1.10:7001,192.168.1.11:7002
```

On Worker 1 (192.168.1.20):

```
mars-worker -a 192.168.1.20 -p 7003 -s 192.168.1.10:7001,192.168.1.11:7002 \
    --spill-dirs /mnt/disk2/spill:/mnt/disk3/spill
```

On Worker 2 (192.168.1.21):

```
mars-worker -a 192.168.1.21 -p 7004 -s 192.168.1.10:7001,192.168.1.11:7002 \
    --spill-dirs /mnt/disk2/spill:/mnt/disk3/spill
```

On the web server (192.168.1.30):

```
mars-web -p 7005 -s 192.168.1.10:7001,192.168.1.11:7002
```

2.2.3 Memory Tuning

Mars worker manages two different parts of memory. The first is private process memory and the second is shared memory between all worker processes handled by plasma_store in Apache Arrow. When Mars Worker starts, it will take 50% of free memory space by default as shared memory and the left as private process memory. What's more, Mars provides soft and hard memory limits for memory allocations, which are 75% and 90% by default. If these configurations does not meet your need, you can configure them when Mars Worker starts. You can use `--cache-mem` argument to configure the size of shared memory, `--phy-mem` to configure total memory size, from which the soft and hard limits are computed.

For instance, by using

```
mars-worker -a localhost -p 9012 -s localhost:9010 --cache-mem 512m --phy-mem 90%
```

We limit the size of shared memory as 512MB and the worker can use up to 90% of total physical memory.

2.3 Overview

Mars tensor is the counterpart of Numpy `numpy.ndarray` and implements a subset of the Numpy ndarray interface. It tiles a large tensor into small chunks and describe the inner computation with a directed graph. This lets us compute on tensors larger than memory and take advantage of the ability of multi-cores or distributed clusters.

The following is a brief overview of supported subset of Numpy interface.

- Arithmetic and mathematics: `+`, `-`, `*`, `/`, `exp`, `log`, etc.
- Reduction along axes (`sum`, `max`, `argmax`, etc).
- Most of the array creation routines (`empty`, `ones_like`, `diag`, etc). What's more, Mars does not only support create array/tensor on GPU, but also support create sparse tensor.
- Most of the array manipulation routines (`reshape`, `rollaxis`, `concatenate`, etc.)
- Basic indexing (indexing by ints, slices, newaxes, and Ellipsis).
- Advanced indexing (except combing boolean array indexing and integer array indexing).
- universal functions for elementwise operations.
- Linear algebra functions, including product (`dot`, `matmul`, etc.) and decomposition (`cholesky`, `svd`, etc.).

However, Mars has not implemented entire Numpy interface, either the time limitation or difficulty is the main handicap. Any contribution from community is sincerely welcomed. The main feature not implemented are listed below:

- Tensor with unknown shape does not support all operations.

- Only small subset of `np.linalg` are implemented.
- Operations like `sort` which is hard to execute in parallel are not implemented.
- Mars tensor doesn't implement interface like `tolist` and `nditer` etc, because the iteration or loops over a large tensor is very inefficient.

2.4 Create Mars tensor

You can create mars tensor from Python array like object just like Numpy, or create from Numpy array directly. More details on *array creation routine* and *random sampling*.

mars.tensor.tensor	
mars.tensor.array	Create a tensor.

2.4.1 mars.tensor.tensor

mars.tensor.**tensor**(*data*, *dtype=None*, *order='K'*, *chunk_size=None*, *gpu=None*, *sparse=False*)

2.4.2 mars.tensor.array

mars.tensor.**array**(*x*, *dtype=None*, *copy=True*, *order='K'*, *ndmin=None*, *chunk_size=None*)

Create a tensor.

object [array_like] An array, any object exposing the array interface, an object whose __array__ method returns an array, or any (nested) sequence.

dtype [data-type, optional] The desired data-type for the array. If not given, then the type will be determined as the minimum type required to hold the objects in the sequence. This argument can only be used to 'upcast' the array. For downcasting, use the .astype(t) method.

copy [bool, optional] If true (default), then the object is copied. Otherwise, a copy will only be made if __array__ returns a copy, if obj is a nested sequence, or if a copy is needed to satisfy any of the other requirements (*dtype*, *order*, etc.).

order [{'K', 'A', 'C', 'F'}, optional] Specify the memory layout of the array. If object is not an array, the newly created array will be in C order (row major) unless 'F' is specified, in which case it will be in Fortran order (column major). If object is an array the following holds.

order	no copy	copy=True
'K'	unchanged	F & C order preserved, otherwise most similar order
'A'	unchanged	F order if input is F and not C, otherwise C order
'C'	C order	C order
'F'	F order	F order

When `copy=False` and a copy is made for other reasons, the result is the same as if `copy=True`, with some exceptions for *A*, see the Notes section. The default order is 'K'.

ndmin [int, optional] Specifies the minimum number of dimensions that the resulting array should have. Ones will be pre-pended to the shape as needed to meet this requirement.

chunk_size: int, tuple, optional Specifies chunk size for each dimension.

out [Tensor] An tensor object satisfying the specified requirements.

empty, empty_like, zeros, zeros_like, ones, ones_like, full, full_like

```
>>> import mars.tensor as mt
```

```
>>> mt.array([1, 2, 3]).execute()
array([1, 2, 3])
```

Upcasting:

```
>>> mt.array([1, 2, 3.0]).execute()
array([ 1.,  2.,  3.])
```

More than one dimension:

```
>>> mt.array([[1, 2], [3, 4]]).execute()
array([[1, 2],
       [3, 4]])
```

Minimum dimensions 2:

```
>>> mt.array([1, 2, 3], ndmin=2).execute()
array([[1, 2, 3]])
```

Type provided:

```
>>> mt.array([1, 2, 3], dtype=complex).execute()
array([ 1.+0.j,  2.+0.j,  3.+0.j])
```

2.4.3 Create tensor on GPU

Mars tensor can run on GPU, for tensor creation, just add a `gpu` parameter, and set it to `True`.

```
import mars.tensor as mt

a = mt.random.rand(1000, 2000, gpu=True)  # allocate the tensor on GPU
```

2.4.4 Create sparse tensor

Mars tensor can be sparse, unfortunately, only 2-D sparse tensors are supported for now, multi-dimensional tensor will be supported later soon.

```
import mars.tensor as mt

a = mt.eye(1000, sparse=True)  # create a sparse 2-D tensor with ones on the diagonal␣
↪and zeros elsewhere
```

2.4.5 Chunks

In mars tensor, we tile a tensor into small chunks. Argument `chunk_size` is not always required, a chunk's bytes occupation will be 128M for the default setting. However, user can specify each chunk's size in a more flexible way

which may be adaptive to the data scale. The fact is that chunk's size may effect heavily on the performance of execution.

The options or arguments which will effect the chunk's size are listed below:

- Change `options.tensor.chunk_size_limit` which is 128*1024*1024(128M) by default.
- Specify `chunk_size` as integer, like `5000`, means chunk's size is 5000 at most for all dimensions
- Specify `chunk_size` as tuple, like `(5000, 3000)`
- Explicitly define sizes of all chunks along all dimensions, like `((5000, 5000, 2000), (2000, 1000))`

Chunks Examples

Assume we have such a tensor with the data shown below.

```
0 9 6 7 6 6
5 7 5 6 9 0
1 6 7 8 6 1
8 0 9 9 9 3
5 4 3 5 8 2
6 2 2 6 9 3
4 2 4 6 2 0
6 8 2 6 5 4
```

We will show how different `chunk_size` arguments will tile the tensor.

`chunk_size=3`:

```
0 9 6  7 6 6
5 7 5  6 9 0
1 6 7  8 6 1

8 0 9  9 9 3
5 4 3  5 8 2
6 2 2  6 9 3

4 2 4  6 2 0
6 8 2  6 5 4
```

`chunk_size=2`:

```
0 9  6 7  6 6
5 7  5 6  9 0

1 6  7 8  6 1
8 0  9 9  9 3

5 4  3 5  8 2
6 2  2 6  9 3

4 2  4 6  2 0
6 8  2 6  5 4
```

`chunk_size=(3, 2)`:

```
0 9  6 7  6 6
5 7  5 6  9 0
1 6  7 8  6 1

8 0  9 9  9 3
5 4  3 5  8 2
6 2  2 6  9 3

4 2  4 6  2 0
6 8  2 6  5 4
```

`chunk_size=((3, 1, 2, 2), (3, 2, 1))`:

```
0 9 6  7 6  6
5 7 5  6 9  0
1 6 7  8 6  1

8 0 9  9 9  3

5 4 3  5 8  2
6 2 2  6 9  3

4 2 4  6 2  0
6 8 2  6 5  4
```

2.5 Universal Functions (ufunc)

Mars tensor provides universal functions(a.k.a ufuncs) to support various elementwise operations. Mars tensor's ufunc supports following features of Numpy's one:

- Broadcasting
- Output type determination
- Casting rules

Mars tensor's ufunc currently does not support methods like `reduce`, `accumulate`, `reduceat`, `outer`, and `at`.

2.5.1 Available ufuncs

Math operations

mars.tensor.add	Add arguments element-wise.
mars.tensor.subtract	Subtract arguments, element-wise.
mars.tensor.multiply	Multiply arguments element-wise.
mars.tensor.divide	Divide arguments element-wise.
mars.tensor.logaddexp	Logarithm of the sum of exponentiations of the inputs.
mars.tensor.logaddexp2	Logarithm of the sum of exponentiations of the inputs in base-2.
mars.tensor.true_divide	Returns a true division of the inputs, element-wise.
mars.tensor.floor_divide	Return the largest integer smaller or equal to the division of the inputs.

Continued on next page

Table 2 – continued from previous page

mars.tensor.negative	Numerical negative, element-wise.
mars.tensor.power	First tensor elements raised to powers from second tensor, element-wise.
mars.tensor.remainder	Return element-wise remainder of division.
mars.tensor.mod	Return element-wise remainder of division.
mars.tensor.fmod	Return the element-wise remainder of division.
mars.tensor.absolute	Calculate the absolute value element-wise.
mars.tensor.rint	Round elements of the tensor to the nearest integer.
mars.tensor.sign	Returns an element-wise indication of the sign of a number.
mars.tensor.exp	Calculate the exponential of all elements in the input tensor.
mars.tensor.exp2	Calculate *2**p* for all *p* in the input tensor.
mars.tensor.log	Natural logarithm, element-wise.
mars.tensor.log2	Base-2 logarithm of *x*.
mars.tensor.log10	Return the base 10 logarithm of the input tensor, element-wise.
mars.tensor.expm1	Calculate `exp(x) - 1` for all elements in the tensor.
mars.tensor.log1p	Return the natural logarithm of one plus the input tensor, element-wise.
mars.tensor.sqrt	Return the positive square-root of an tensor, element-wise.
mars.tensor.square	Return the element-wise square of the input.
mars.tensor.reciprocal	Return the reciprocal of the argument, element-wise.

mars.tensor.add

`mars.tensor.`**`add`**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Add arguments element-wise.

x1, x2 [array_like] The tensors to be added. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which may be the shape of one or the other).

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

add [Tensor or scalar] The sum of *x1* and *x2*, element-wise. Returns a scalar if both *x1* and *x2* are scalars.

Equivalent to *x1* + *x2* in terms of tensor broadcasting.

```
>>> import mars.tensor as mt
```

```
>>> mt.add(1.0, 4.0).execute()
5.0
>>> x1 = mt.arange(9.0).reshape((3, 3))
>>> x2 = mt.arange(3.0)
>>> mt.add(x1, x2).execute()
array([[  0.,   2.,   4.],
```

(continues on next page)

(continued from previous page)

```
       [  3.,   5.,   7.],
       [  6.,   8.,  10.]])
```

mars.tensor.subtract

mars.tensor.**subtract**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Subtract arguments, element-wise.

x1, x2 [array_like] The tensors to be subtracted from each other.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y [Tensor] The difference of *x1* and *x2*, element-wise. Returns a scalar if both *x1* and *x2* are scalars.

Equivalent to `x1 - x2` in terms of tensor broadcasting.

```
>>> import mars.tensor as mt
```

```
>>> mt.subtract(1.0, 4.0).execute()
-3.0
```

```
>>> x1 = mt.arange(9.0).reshape((3, 3))
>>> x2 = mt.arange(3.0)
>>> mt.subtract(x1, x2).execute()
array([[ 0.,  0.,  0.],
       [ 3.,  3.,  3.],
       [ 6.,  6.,  6.]])
```

mars.tensor.multiply

mars.tensor.**multiply**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Multiply arguments element-wise.

x1, x2 [array_like] Input arrays to be multiplied.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y [Tensor] The product of *x1* and *x2*, element-wise. Returns a scalar if both *x1* and *x2* are scalars.

Equivalent to *x1 * x2* in terms of array broadcasting.

```
>>> import mars.tensor as mt
```

```
>>> mt.multiply(2.0, 4.0).execute()
8.0
```

```
>>> x1 = mt.arange(9.0).reshape((3, 3))
>>> x2 = mt.arange(3.0)
>>> mt.multiply(x1, x2).execute()
array([[  0.,   1.,   4.],
       [  0.,   4.,  10.],
       [  0.,   7.,  16.]])
```

mars.tensor.divide

mars.tensor.**divide**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Divide arguments element-wise.

x1 [array_like] Dividend tensor.

x2 [array_like] Divisor tensor.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

out [Tensor] The quotient *x1/x2*, element-wise. Returns a scalar if both *x1* and *x2* are scalars.

Equivalent to *x1 / x2* in terms of array-broadcasting.

Behavior on division by zero can be changed using *seterr*.

In Python 2, when both *x1* and *x2* are of an integer type, *divide* will behave like *floor_divide*. In Python 3, it behaves like *true_divide*.

```
>>> import mars.tensor as mt
```

```
>>> mt.divide(2.0, 4.0).execute()
0.5
>>> x1 = mt.arange(9.0).reshape((3, 3))
>>> x2 = mt.arange(3.0)
>>> mt.divide(x1, x2).execute()
array([[ NaN,  1. ,  1. ],
       [ Inf,  4. ,  2.5],
       [ Inf,  7. ,  4. ]])
Note the behavior with integer types (Python 2 only):
>>> mt.divide(2, 4).execute()
0
>>> mt.divide(2, 4.).execute()
0.5
Division by zero always yields zero in integer arithmetic (again, Python 2 only),
and does not raise an exception or a warning:
>>> mt.divide(mt.array([0, 1], dtype=int), mt.array([0, 0], dtype=int)).execute()
```

(continues on next page)

(continued from previous page)

```
array([0, 0])
Division by zero can, however, be caught using seterr:
>>> old_err_state = mt.seterr(divide='raise')
>>> mt.divide(1, 0).execute()
Traceback (most recent call last):
...
FloatingPointError: divide by zero encountered in divide
>>> ignored_states = mt.seterr(**old_err_state)
>>> mt.divide(1, 0).execute()
0
```

mars.tensor.logaddexp

mars.tensor.**logaddexp**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Logarithm of the sum of exponentiations of the inputs.

Calculates `log(exp(x1) + exp(x2))`. This function is useful in statistics where the calculated probabilities of events may be so small as to exceed the range of normal floating point numbers. In such cases the logarithm of the calculated probability is stored. This function allows adding probabilities stored in such a fashion.

x1, x2 [array_like] Input values.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs** For other keyword-only arguments, see the ufunc docs.

result [Tensor] Logarithm of `exp(x1) + exp(x2)`.

logaddexp2: Logarithm of the sum of exponentiations of inputs in base 2.

```
>>> import mars.tensor as mt
```

```
>>> prob1 = mt.log(1e-50)
>>> prob2 = mt.log(2.5e-50)
>>> prob12 = mt.logaddexp(prob1, prob2)
>>> prob12.execute()
-113.87649168120691
>>> mt.exp(prob12).execute()
3.5000000000000057e-50
```

mars.tensor.logaddexp2

mars.tensor.**logaddexp2**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Logarithm of the sum of exponentiations of the inputs in base-2.

Calculates `log2(2**x1 + 2**x2)`. This function is useful in machine learning when the calculated probabilities of events may be so small as to exceed the range of normal floating point numbers. In such cases the

base-2 logarithm of the calculated probability can be used instead. This function allows adding probabilities stored in such a fashion.

x1, x2 [array_like] Input values.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

result [Tensor] Base-2 logarithm of `2**x1 + 2**x2`.

logaddexp: Logarithm of the sum of exponentiations of the inputs.

```
>>> import mars.tensor as mt
```

```
>>> prob1 = mt.log2(1e-50)
>>> prob2 = mt.log2(2.5e-50)
>>> prob12 = mt.logaddexp2(prob1, prob2)
>>> prob1.execute(), prob2.execute(), prob12.execute()
(-166.09640474436813, -164.77447664948076, -164.28904982231052)
>>> (2**prob12).execute()
3.4999999999999914e-50
```

mars.tensor.true_divide

mars.tensor.**true_divide**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Returns a true division of the inputs, element-wise.

Instead of the Python traditional 'floor division', this returns a true division. True division adjusts the output type to present the best answer, regardless of input types.

x1 [array_like] Dividend tensor.

x2 [array_like] Divisor tensor.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

out [Tensor] Result is scalar if both inputs are scalar, tensor otherwise.

The floor division operator `//` was added in Python 2.2 making `//` and `/` equivalent operators. The default floor division operation of `/` can be replaced by true division with `from __future__ import division`.

In Python 3.0, `//` is the floor division operator and `/` the true division operator. The `true_divide(x1, x2)` function is equivalent to true division in Python.

```
>>> import mars.tensor as mt
```

```
>>> x = mt.arange(5)
>>> mt.true_divide(x, 4).execute()
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])
```

for python 2 >>> (x/4).execute() array([0, 0, 0, 0, 1]) >>> (x//4).execute() array([0, 0, 0, 0, 1])

mars.tensor.floor_divide

mars.tensor.**floor_divide**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Return the largest integer smaller or equal to the division of the inputs. It is equivalent to the Python `//` operator and pairs with the Python `%` (*remainder*), function so that `b = a % b + b * (a // b)` up to roundoff.

x1 [array_like] Numerator.

x2 [array_like] Denominator.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y [Tensor] y = floor(*x1*/*x2*)

remainder : Remainder complementary to floor_divide. divmod : Simultaneous floor division and remainder. divide : Standard division. floor : Round a number to the nearest integer toward minus infinity. ceil : Round a number to the nearest integer toward infinity.

```
>>> import mars.tensor as mt
```

```
>>> mt.floor_divide(7,3).execute()
2
>>> mt.floor_divide([1., 2., 3., 4.], 2.5).execute()
array([ 0.,  0.,  1.,  1.])
```

mars.tensor.negative

mars.tensor.**negative**(*x*, *out=None*, *where=None*, ***kwargs*)

Numerical negative, element-wise.

x [array_like or scalar] Input tensor.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs** For other keyword-only arguments, see the ufunc docs.

y [Tensor or scalar] Returned array or scalar: *y* = *-x*.

```
>>> import mars.tensor as mt
```

```
>>> mt.negative([1.,-1.]).execute()
array([-1.,  1.])
```

mars.tensor.power

mars.tensor.**power**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

First tensor elements raised to powers from second tensor, element-wise.

Raise each base in *x1* to the positionally-corresponding power in *x2*. *x1* and *x2* must be broadcastable to the same shape. Note that an integer type raised to a negative integer power will raise a ValueError.

x1 [array_like] The bases.

x2 [array_like] The exponents.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y [Tensor] The bases in *x1* raised to the exponents in *x2*.

float_power : power function that promotes integers to float

Cube each element in a list.

```
>>> import mars.tensor as mt
```

```
>>> x1 = range(6)
>>> x1
[0, 1, 2, 3, 4, 5]
>>> mt.power(x1, 3).execute()
array([  0,   1,   8,  27,  64, 125])
```

Raise the bases to different exponents.

```
>>> x2 = [1.0, 2.0, 3.0, 3.0, 2.0, 1.0]
>>> mt.power(x1, x2).execute()
array([  0.,   1.,   8.,  27.,  16.,   5.])
```

The effect of broadcasting.

```
>>> x2 = mt.array([[1, 2, 3, 3, 2, 1], [1, 2, 3, 3, 2, 1]])
>>> x2.execute()
array([[1, 2, 3, 3, 2, 1],
       [1, 2, 3, 3, 2, 1]])
>>> mt.power(x1, x2).execute()
array([[ 0,  1,  8, 27, 16,  5],
       [ 0,  1,  8, 27, 16,  5]])
```

mars.tensor.remainder

mars.tensor.**remainder**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Return element-wise remainder of division.

Computes the remainder complementary to the *floor_divide* function. It is equivalent to the Python modulus operator``x1 % x2`` and has the same sign as the divisor *x2*. The MATLAB function equivalent to `np.remainder` is `mod`.

> **Warning:** This should not be confused with:
>
> - Python 3.7's *math.remainder* and C's `remainder`, which computes the IEEE remainder, which are the complement to `round(x1 / x2)`.
> - The MATLAB `rem` function and or the C `%` operator which is the complement to `int(x1 / x2)`.

x1 [array_like] Dividend array.

x2 [array_like] Divisor array.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y [Tensor] The element-wise remainder of the quotient `floor_divide(x1, x2)`. Returns a scalar if both *x1* and *x2* are scalars.

floor_divide : Equivalent of Python `//` operator. divmod : Simultaneous floor division and remainder. fmod : Equivalent of the MATLAB `rem` function. divide, floor

Returns 0 when *x2* is 0 and both *x1* and *x2* are (tensors of) integers.

```
>>> import mars.tensor as mt
```

```
>>> mt.remainder([4, 7], [2, 3]).execute()
array([0, 1])
>>> mt.remainder(mt.arange(7), 5).execute()
array([0, 1, 2, 3, 4, 0, 1])
```

mars.tensor.mod

mars.tensor.**mod**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Return element-wise remainder of division.

Computes the remainder complementary to the *floor_divide* function. It is equivalent to the Python modulus operator``x1 % x2`` and has the same sign as the divisor *x2*. The MATLAB function equivalent to `np.remainder` is `mod`.

> **Warning:** This should not be confused with:

- Python 3.7's *math.remainder* and C's `remainder`, which computes the IEEE remainder, which are the complement to `round(x1 / x2)`.
- The MATLAB `rem` function and or the C `%` operator which is the complement to `int(x1 / x2)`.

x1 [array_like] Dividend array.

x2 [array_like] Divisor array.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs

y [Tensor] The element-wise remainder of the quotient `floor_divide(x1, x2)`. Returns a scalar if both *x1* and *x2* are scalars.

floor_divide : Equivalent of Python `//` operator. divmod : Simultaneous floor division and remainder. fmod : Equivalent of the MATLAB `rem` function. divide, floor

Returns 0 when *x2* is 0 and both *x1* and *x2* are (tensors of) integers.

```
>>> import mars.tensor as mt
```

```
>>> mt.remainder([4, 7], [2, 3]).execute()
array([0, 1])
>>> mt.remainder(mt.arange(7), 5).execute()
array([0, 1, 2, 3, 4, 0, 1])
```

mars.tensor.fmod

`mars.tensor.`**`fmod`**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Return the element-wise remainder of division.

This is the NumPy implementation of the C library function fmod, the remainder has the same sign as the dividend *x1*. It is equivalent to the Matlab(TM) `rem` function and should not be confused with the Python modulus operator `x1 % x2`.

x1 [array_like] Dividend.

x2 [array_like] Divisor.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs** For other keyword-only arguments, see the ufunc docs.

y [Tensor_like] The remainder of the division of *x1* by *x2*.

remainder : Equivalent to the Python `%` operator. divide

The result of the modulo operation for negative dividend and divisors is bound by conventions. For *fmod*, the sign of result is the sign of the dividend, while for *remainder* the sign of the result is the sign of the divisor. The *fmod* function is equivalent to the Matlab(TM) `rem` function.

```
>>> import mars.tensor as mt
```

```
>>> mt.fmod([-3, -2, -1, 1, 2, 3], 2).execute()
array([-1,  0, -1,  1,  0,  1])
>>> mt.remainder([-3, -2, -1, 1, 2, 3], 2).execute()
array([1, 0, 1, 1, 0, 1])
```

```
>>> mt.fmod([5, 3], [2, 2.]).execute()
array([ 1.,  1.])
>>> a = mt.arange(-3, 3).reshape(3, 2)
>>> a.execute()
array([[-3, -2],
       [-1,  0],
       [ 1,  2]])
>>> mt.fmod(a, [2,2]).execute()
array([[-1,  0],
       [-1,  0],
       [ 1,  0]])
```

mars.tensor.absolute

`mars.tensor.`**`absolute`**(*x*, *out=None*, *where=None*, ***kwargs*)

Calculate the absolute value element-wise.

x [array_like] Input tensor.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

absolute [Tensor] An tensor containing the absolute value of each element in *x*. For complex input, `a + ib`, the absolute value is $\sqrt{a^2 + b^2}$.

```
>>> import mars.tensor as mt
```

```
>>> x = mt.array([-1.2, 1.2])
>>> mt.absolute(x).execute()
array([ 1.2,  1.2])
>>> mt.absolute(1.2 + 1j).execute()
1.5620499351813308
```

mars.tensor.rint

`mars.tensor.`**`rint`**(*x*, *out=None*, *where=None*, ***kwargs*)

Round elements of the tensor to the nearest integer.

x [array_like] Input tensor.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

out [Tensor or scalar] Output array is same shape and type as *x*.

ceil, floor, trunc

```
>>> import mars.tensor as mt
```

```
>>> a = mt.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> mt.rint(a).execute()
array([-2., -2., -0.,  0.,  2.,  2.,  2.])
```

mars.tensor.sign

mars.tensor.**sign**(*x*, *out=None*, *where=None*, ***kwargs*)

Returns an element-wise indication of the sign of a number.

The *sign* function returns `-1 if x < 0, 0 if x==0, 1 if x > 0`. nan is returned for nan inputs.

For complex inputs, the *sign* function returns `sign(x.real) + 0j if x.real != 0 else sign(x.imag) + 0j`.

complex(nan, 0) is returned for complex nan inputs.

x [array_like] Input values.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

y [Tensor] The sign of *x*.

There is more than one definition of sign in common use for complex numbers. The definition used here is equivalent to $x/\sqrt{x*x}$ which is different from a common alternative, $x/|x|$.

```
>>> import mars.tensor as mt
```

```
>>> mt.sign([-5., 4.5]).execute()
array([-1.,  1.])
>>> mt.sign(0).execute()
0
>>> mt.sign(5-2j).execute()
(1+0j)
```

mars.tensor.exp

mars.tensor.**exp**(*x*, *out=None*, *where=None*, ***kwargs*)

Calculate the exponential of all elements in the input tensor.

x [array_like] Input values.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs** For other keyword-only arguments, see the ufunc docs.

out [Tensor] Output tensor, element-wise exponential of *x*.

expm1 : Calculate `exp(x) - 1` for all elements in the array. exp2 : Calculate `2**x` for all elements in the array.

The irrational number `e` is also known as Euler's number. It is approximately 2.718281, and is the base of the natural logarithm, `ln` (this means that, if $x = \ln y = \log_e y$, then $e^x = y$. For real input, `exp(x)` is always positive.

For complex arguments, `x = a + ib`, we can write $e^x = e^a e^{ib}$. The first term, e^a, is already known (it is the real argument, described above). The second term, e^{ib}, is $\cos b + i \sin b$, a function with magnitude 1 and a periodic phase.

Plot the magnitude and phase of `exp(x)` in the complex plane:

```
>>> import mars.tensor as mt
>>> import matplotlib.pyplot as plt
```

```
>>> x = mt.linspace(-2*mt.pi, 2*mt.pi, 100)
>>> xx = x + 1j * x[:, mt.newaxis] # a + ib over complex plane
>>> out = mt.exp(xx)
```

```
>>> plt.subplot(121)
>>> plt.imshow(mt.abs(out).execute(),
...            extent=[-2*mt.pi, 2*mt.pi, -2*mt.pi, 2*mt.pi], cmap='gray')
>>> plt.title('Magnitude of exp(x)')
```

```
>>> plt.subplot(122)
>>> plt.imshow(mt.angle(out).execute(),
...            extent=[-2*mt.pi, 2*mt.pi, -2*mt.pi, 2*mt.pi], cmap='hsv')
>>> plt.title('Phase (angle) of exp(x)')
>>> plt.show()
```

mars.tensor.exp2

mars.tensor.**exp2**(*x*, *out=None*, *where=None*, ***kwargs*)

Calculate *2**p* for all *p* in the input tensor.

x [array_like] Input values.

out [Tensor, None, or tuple of tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

out [Tensor] Element-wise 2 to the power *x*.

power

```
>>> import mars.tensor as mt
```

```
>>> mt.exp2([2, 3]).execute()
array([ 4.,  8.])
```

mars.tensor.log

`mars.tensor.log`(*x*, *out=None*, *where=None*, ***kwargs*)

Natural logarithm, element-wise.

The natural logarithm *log* is the inverse of the exponential function, so that *log(exp(x)) = x*. The natural logarithm is logarithm in base *e*.

x [array_like] Input value.

out [Tensor, None, or tuple of tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

y [Tensor] The natural logarithm of *x*, element-wise.

log10, log2, log1p

Logarithm is a multivalued function: for each *x* there is an infinite number of *z* such that *exp(z) = x*. The convention is to return the *z* whose imaginary part lies in *[-pi, pi]*.

For real-valued input data types, *log* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *log* is a complex analytical function that has a branch cut *[-inf, 0]* and is continuous from above on it. *log* handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

```
>>> import mars.tensor as mt
```

```
>>> mt.log([1, mt.e, mt.e**2, 0]).execute()
array([  0.,   1.,   2., -Inf])
```

mars.tensor.log2

mars.tensor.**log2**(*x*, *out=None*, *where=None*, ***kwargs*)

Base-2 logarithm of *x*.

x [array_like] Input values.

out [Tensor, None, or tuple of tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y [Tensor] Base-2 logarithm of *x*.

log, log10, log1p

Logarithm is a multivalued function: for each *x* there is an infinite number of *z* such that *2**z = x*. The convention is to return the *z* whose imaginary part lies in *[-pi, pi]*.

For real-valued input data types, *log2* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *log2* is a complex analytical function that has a branch cut *[-inf, 0]* and is continuous from above on it. *log2* handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

```
>>> import mars.tensor as mt
```

```
>>> x = mt.array([0, 1, 2, 2**4])
>>> mt.log2(x).execute()
array([-Inf,   0.,   1.,   4.])
```

```
>>> xi = mt.array([0+1.j, 1, 2+0.j, 4.j])
>>> mt.log2(xi).execute()
array([ 0.+2.26618007j,  0.+0.j        ,  1.+0.j        ,  2.+2.26618007j])
```

mars.tensor.log10

mars.tensor.**log10**(*x*, *out=None*, *where=None*, ***kwargs*)

Return the base 10 logarithm of the input tensor, element-wise.

x [array_like] Input values.

out [Tensor, None, or tuple of tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y [Tensor] The logarithm to the base 10 of *x*, element-wise. NaNs are returned where x is negative.

Logarithm is a multivalued function: for each *x* there is an infinite number of *z* such that *10**z = x*. The convention is to return the *z* whose imaginary part lies in *[-pi, pi]*.

For real-valued input data types, *log10* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *log10* is a complex analytical function that has a branch cut *[-inf, 0]* and is continuous from above on it. *log10* handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

```
>>> import mars.tensor as mt
```

```
>>> mt.log10([1e-15, -3.]).execute()
array([-15.,  NaN])
```

mars.tensor.expm1

`mars.tensor.`**`expm1`**(*x*, *out=None*, *where=None*, ***kwargs*)

Calculate `exp(x) - 1` for all elements in the tensor.

x [array_like] Input values.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

out [Tensor] Element-wise exponential minus one: `out = exp(x) - 1`.

log1p : `log(1 + x)`, the inverse of expm1.

This function provides greater precision than `exp(x) - 1` for small values of `x`.

The true value of `exp(1e-10) - 1` is `1.00000000005e-10` to about 32 significant digits. This example shows the superiority of expm1 in this case.

```
>>> import mars.tensor as mt
```

```
>>> mt.expm1(1e-10).execute()
1.00000000005e-10
>>> (mt.exp(1e-10) - 1).execute()
1.000000082740371e-10
```

mars.tensor.log1p

`mars.tensor.`**`log1p`**(*x*, *out=None*, *where=None*, ***kwargs*)

Return the natural logarithm of one plus the input tensor, element-wise.

Calculates `log(1 + x)`.

x [array_like] Input values.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

y [Tensor] Natural logarithm of *1 + x*, element-wise.

expm1 : `exp(x) - 1`, the inverse of *log1p*.

For real-valued input, *log1p* is accurate also for *x* so small that *1 + x == 1* in floating-point accuracy.

Logarithm is a multivalued function: for each *x* there is an infinite number of *z* such that *exp(z) = 1 + x*. The convention is to return the *z* whose imaginary part lies in *[-pi, pi]*.

For real-valued input data types, *log1p* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *log1p* is a complex analytical function that has a branch cut *[-inf, -1]* and is continuous from above on it. *log1p* handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

```
>>> import mars.tensor as mt
```

```
>>> mt.log1p(1e-99).execute()
1e-99
>>> mt.log(1 + 1e-99).execute()
0.0
```

mars.tensor.sqrt

`mars.tensor.`**`sqrt`**(*x*, *out=None*, *where=None*, ***kwargs*)

Return the positive square-root of an tensor, element-wise.

x [array_like] The values whose square-roots are required.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

y [Tensor] An tensor of the same shape as *x*, containing the positive square-root of each element in *x*. If any element in *x* is complex, a complex tensor is returned (and the square-roots of negative reals are calculated). If all of the elements in *x* are real, so is *y*, with negative elements returning `nan`. If *out* was provided, *y* is a reference to it.

sqrt has–consistent with common convention–as its branch cut the real "interval" [*-inf*, 0), and is continuous from above on it. A branch cut is a curve in the complex plane across which a given complex function fails to be continuous.

```
>>> import mars.tensor as mt
```

```
>>> mt.sqrt([1,4,9]).execute()
array([ 1.,  2.,  3.])
```

```
>>> mt.sqrt([4, -1, -3+4J]).execute()
array([ 2.+0.j,  0.+1.j,  1.+2.j])
```

```
>>> mt.sqrt([4, -1, mt.inf]).execute()
array([  2.,  NaN,  Inf])
```

mars.tensor.square

mars.tensor.**square**(*x*, *out=None*, *where=None*, ***kwargs*)

Return the element-wise square of the input.

x [array_like] Input data.

out [Tensor, None, or tuple of tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

out [Tensor] Element-wise *x*x*, of the same shape and dtype as *x*. Returns scalar if *x* is a scalar.

sqrt power

```
>>> import mars.tensor as mt
```

```
>>> mt.square([-1j, 1]).execute()
array([-1.-0.j,  1.+0.j])
```

mars.tensor.reciprocal

mars.tensor.**reciprocal**(*x*, *out=None*, *where=None*, ***kwargs*)

Return the reciprocal of the argument, element-wise.

Calculates `1/x`.

x [array_like] Input tensor.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y [Tensor] Return tensor.

Note: This function is not designed to work with integers.

For integer arguments with absolute value larger than 1 the result is always zero because of the way Python handles integer division. For integer zero the result is an overflow.

```
>>> import mars.tensor as mt
```

```
>>> mt.reciprocal(2.).execute()
0.5
>>> mt.reciprocal([1, 2., 3.33]).execute()
array([ 1.       ,  0.5      ,  0.3003003])
```

Trigonometric functions

mars.tensor.sin	Trigonometric sine, element-wise.
mars.tensor.cos	Cosine element-wise.
mars.tensor.tan	Compute tangent element-wise.
mars.tensor.arcsin	Inverse sine, element-wise.
mars.tensor.arccos	Trigonometric inverse cosine, element-wise.
mars.tensor.arctan	Trigonometric inverse tangent, element-wise.
mars.tensor.arctan2	Element-wise arc tangent of `x1/x2` choosing the quadrant correctly.
mars.tensor.hypot	Given the "legs" of a right triangle, return its hypotenuse.
mars.tensor.sinh	Hyperbolic sine, element-wise.
mars.tensor.cosh	Hyperbolic cosine, element-wise.
mars.tensor.tanh	Compute hyperbolic tangent element-wise.
mars.tensor.arcsinh	Inverse hyperbolic sine element-wise.
mars.tensor.arccosh	Inverse hyperbolic cosine, element-wise.
mars.tensor.arctanh	Inverse hyperbolic tangent element-wise.
mars.tensor.deg2rad	Convert angles from degrees to radians.
mars.tensor.rad2deg	Convert angles from radians to degrees.

mars.tensor.sin

mars.tensor.**sin**(*x*, *out=None*, *where=None*, ***kwargs*)

Trigonometric sine, element-wise.

x [array_like] Angle, in radians (2π rad equals 360 degrees).

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y [array_like] The sine of each element of x.

arcsin, sinh, cos

The sine is one of the fundamental functions of trigonometry (the mathematical study of triangles). Consider a circle of radius 1 centered on the origin. A ray comes in from the $+x$ axis, makes an angle at the origin (measured counter-clockwise from that axis), and departs from the origin. The y coordinate of the outgoing

ray's intersection with the unit circle is the sine of that angle. It ranges from -1 for $x = 3\pi/2$ to +1 for $\pi/2$. The function has zeroes where the angle is a multiple of π. Sines of angles between π and 2π are negative. The numerous properties of the sine and related functions are included in any standard trigonometry text.

Print sine of one angle:

```
>>> import mars.tensor as mt
```

```
>>> mt.sin(mt.pi/2.).execute()
1.0
```

Print sines of an array of angles given in degrees:

```
>>> mt.sin(mt.array((0., 30., 45., 60., 90.)) * mt.pi / 180. ).execute()
array([ 0.        ,  0.5       ,  0.70710678,  0.8660254 ,  1.        ])
```

Plot the sine function:

```
>>> import matplotlib.pylab as plt
>>> x = mt.linspace(-mt.pi, mt.pi, 201)
>>> plt.plot(x.execute(), mt.sin(x).execute())
>>> plt.xlabel('Angle [rad]')
>>> plt.ylabel('sin(x)')
>>> plt.axis('tight')
>>> plt.show()
```

mars.tensor.cos

mars.tensor.**cos**(*x*, *out=None*, *where=None*, ***kwargs*)

Cosine element-wise.

x [array_like] Input tensor in radians.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y [Tensor] The corresponding cosine values.

If *out* is provided, the function writes the result into it, and returns a reference to *out*. (See Examples)

M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972.

```
>>> import mars.tensor as mt
```

```
>>> mt.cos(mt.array([0, mt.pi/2, mt.pi])).execute()
array([  1.00000000e+00,   6.12303177e-17,  -1.00000000e+00])
>>>
>>> # Example of providing the optional output parameter
>>> out1 = mt.empty(1)
>>> out2 = mt.cos([0.1], out1)
>>> out2 is out1
True
```

(continues on next page)

(continued from previous page)

```
>>>
>>> # Example of ValueError due to provision of shape mis-matched `out`
>>> mt.cos(mt.zeros((3,3)),mt.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (3,3) (2,2)
```

mars.tensor.tan

mars.tensor.**tan**(*x*, *out=None*, *where=None*, ***kwargs*)

Compute tangent element-wise.

Equivalent to `mt.sin(x)/mt.cos(x)` element-wise.

x [array_like] Input tensor.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y [Tensor] The corresponding tangent values.

If *out* is provided, the function writes the result into it, and returns a reference to *out*. (See Examples)

M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972.

```
>>> from math import pi
>>> import mars.tensor as mt
>>> mt.tan(mt.array([-pi,pi/2,pi])).execute()
array([  1.22460635e-16,   1.63317787e+16,  -1.22460635e-16])
>>>
>>> # Example of providing the optional output parameter illustrating
>>> # that what is returned is a reference to said parameter
>>> out1 = mt.zeros(1)
>>> out2 = mt.cos([0.1], out1)
>>> out2 is out1
True
>>>
>>> # Example of ValueError due to provision of shape mis-matched `out`
>>> mt.cos(mt.zeros((3,3)),mt.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid return array shape
```

mars.tensor.arcsin

mars.tensor.**arcsin**(*x*, *out=None*, *where=None*, ***kwargs*)

Inverse sine, element-wise.

x [array_like] *y*-coordinate on the unit circle.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

angle [Tensor] The inverse sine of each element in *x*, in radians and in the closed interval `[-pi/2, pi/2]`. If *x* is a scalar, a scalar is returned, otherwise a tensor.

sin, cos, arccos, tan, arctan, arctan2, emath.arcsin

arcsin is a multivalued function: for each *x* there are infinitely many numbers *z* such that $sin(z) = x$. The convention is to return the angle *z* whose real part lies in [-pi/2, pi/2].

For real-valued input data types, *arcsin* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *arcsin* is a complex analytic function that has, by convention, the branch cuts [-inf, -1] and [1, inf] and is continuous from above on the former and from below on the latter.

The inverse sine is also known as *asin* or sin^{-1}.

Abramowitz, M. and Stegun, I. A., *Handbook of Mathematical Functions*, 10th printing, New York: Dover, 1964, pp. 79ff. http://www.math.sfu.ca/~cbm/aands/

```
>>> import mars.tensor as mt
>>> mt.arcsin(1).execute()      # pi/2
1.5707963267948966
>>> mt.arcsin(-1).execute()     # -pi/2
-1.5707963267948966
>>> mt.arcsin(0).execute()
0.0
```

mars.tensor.arccos

`mars.tensor.`**`arccos`**(*x*, *out=None*, *where=None*, ***kwargs*)

Trigonometric inverse cosine, element-wise.

The inverse of *cos* so that, if `y = cos(x)`, then `x = arccos(y)`.

x [array_like] *x*-coordinate on the unit circle. For real arguments, the domain is [-1, 1].

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

angle [Tensor] The angle of the ray intersecting the unit circle at the given *x*-coordinate in radians [0, pi]. If *x* is a scalar then a scalar is returned, otherwise an array of the same shape as *x* is returned.

cos, arctan, arcsin

arccos is a multivalued function: for each *x* there are infinitely many numbers *z* such that *cos(z) = x*. The convention is to return the angle *z* whose real part lies in *[0, pi]*.

For real-valued input data types, *arccos* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *arccos* is a complex analytic function that has branch cuts *[-inf, -1]* and *[1, inf]* and is continuous from above on the former and from below on the latter.

The inverse *cos* is also known as *acos* or cos^-1.

M. Abramowitz and I.A. Stegun, "Handbook of Mathematical Functions", 10th printing, 1964, pp. 79. http://www.math.sfu.ca/~cbm/aands/

We expect the arccos of 1 to be 0, and of -1 to be pi: >>> import mars.tensor as mt

```
>>> mt.arccos([1, -1]).execute()
array([ 0.        ,  3.14159265])
```

Plot arccos:

```
>>> import matplotlib.pyplot as plt
>>> x = mt.linspace(-1, 1, num=100)
>>> plt.plot(x.execute(), mt.arccos(x).execute())
>>> plt.axis('tight')
>>> plt.show()
```

mars.tensor.arctan

`mars.tensor.`**`arctan`**(*x*, *out=None*, *where=None*, ***kwargs*)

Trigonometric inverse tangent, element-wise.

The inverse of tan, so that if `y = tan(x)` then `x = arctan(y)`.

x : array_like out : Tensor, None, or tuple of Tensor and None, optional

A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

out [Tensor] Out has the same shape as *x*. Its real part is in `[-pi/2, pi/2]` (`arctan(+/-inf)` returns `+/-pi/2`). It is a scalar if *x* is a scalar.

arctan2 [The "four quadrant" arctan of the angle formed by (*x*, *y*)] and the positive *x*-axis.

angle : Argument of complex values.

arctan is a multi-valued function: for each *x* there are infinitely many numbers *z* such that tan(*z*) = *x*. The convention is to return the angle *z* whose real part lies in [-pi/2, pi/2].

For real-valued input data types, *arctan* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *arctan* is a complex analytic function that has [*1j, infj*] and [*-1j, -infj*] as branch cuts, and is continuous from the left on the former and from the right on the latter.

The inverse tangent is also known as *atan* or tan^{-1}.

Abramowitz, M. and Stegun, I. A., *Handbook of Mathematical Functions*, 10th printing, New York: Dover, 1964, pp. 79. http://www.math.sfu.ca/~cbm/aands/

We expect the arctan of 0 to be 0, and of 1 to be pi/4: >>> import mars.tensor as mt

```
>>> mt.arctan([0, 1]).execute()
array([ 0.        ,  0.78539816])
```

```
>>> mt.pi/4
0.78539816339744828
```

Plot arctan:

```
>>> import matplotlib.pyplot as plt
>>> x = mt.linspace(-10, 10)
>>> plt.plot(x.execute(), mt.arctan(x).execute())
>>> plt.axis('tight')
>>> plt.show()
```

mars.tensor.arctan2

mars.tensor.**arctan2**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Element-wise arc tangent of `x1/x2` choosing the quadrant correctly.

The quadrant (i.e., branch) is chosen so that `arctan2(x1, x2)` is the signed angle in radians between the ray ending at the origin and passing through the point (1,0), and the ray ending at the origin and passing through the point (*x2*, *x1*). (Note the role reversal: the "*y*-coordinate" is the first function parameter, the "*x*-coordinate" is the second.) By IEEE convention, this function is defined for *x2* = +/-0 and for either or both of *x1* and *x2* = +/-inf (see Notes for specific values).

This function is not defined for complex-valued arguments; for the so-called argument of complex values, use *angle*.

x1 [array_like, real-valued] *y*-coordinates.

x2 [array_like, real-valued] *x*-coordinates. *x2* must be broadcastable to match the shape of *x1* or vice versa.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

angle [Tensor] Array of angles in radians, in the range `[-pi, pi]`.

arctan, tan, angle

arctan2 is identical to the *atan2* function of the underlying C library. The following special values are defined in the C standard:[1]

[1] ISO/IEC standard 9899:1999, "Programming language C."

x1	*x2*	*arctan2(x1,x2)*
+/- 0	+0	+/- 0
+/- 0	-0	+/- pi
> 0	+/-inf	+0 / +pi
< 0	+/-inf	-0 / -pi
+/-inf	+inf	+/- (pi/4)
+/-inf	-inf	+/- (3*pi/4)

Note that +0 and -0 are distinct floating point numbers, as are +inf and -inf.

Consider four points in different quadrants: >>> import mars.tensor as mt

```
>>> x = mt.array([-1, +1, +1, -1])
>>> y = mt.array([-1, -1, +1, +1])
>>> (mt.arctan2(y, x) * 180 / mt.pi).execute()
array([-135.,  -45.,   45.,  135.])
```

Note the order of the parameters. *arctan2* is defined also when *x2* = 0 and at several other special points, obtaining values in the range `[-pi, pi]`:

```
>>> mt.arctan2([1., -1.], [0., 0.]).execute()
array([ 1.57079633, -1.57079633])
>>> mt.arctan2([0., 0., mt.inf], [+0., -0., mt.inf]).execute()
array([ 0.        ,  3.14159265,  0.78539816])
```

mars.tensor.hypot

`mars.tensor.`**`hypot`**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Given the "legs" of a right triangle, return its hypotenuse.

Equivalent to `sqrt(x1**2 + x2**2)`, element-wise. If *x1* or *x2* is scalar_like (i.e., unambiguously castable to a scalar type), it is broadcast for use with each element of the other argument. (See Examples)

x1, x2 [array_like] Leg of the triangle(s).

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

z [Tensor] The hypotenuse of the triangle(s).

```
>>> import mars.tensor as mt
```

```
>>> mt.hypot(3*mt.ones((3, 3)), 4*mt.ones((3, 3))).execute()
array([[ 5.,  5.,  5.],
       [ 5.,  5.,  5.],
       [ 5.,  5.,  5.]])
```

Example showing broadcast of scalar_like argument:

```
>>> mt.hypot(3*mt.ones((3, 3)), [4]).execute()
array([[ 5.,  5.,  5.],
       [ 5.,  5.,  5.],
       [ 5.,  5.,  5.]])
```

mars.tensor.sinh

mars.tensor.**sinh**(*x*, *out=None*, *where=None*, ***kwargs*)

Hyperbolic sine, element-wise.

Equivalent to `1/2 * (mt.exp(x) - mt.exp(-x))` or `-1j * mt.sin(1j*x)`.

x [array_like] Input tensor.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y [Tensor] The corresponding hyperbolic sine values.

If *out* is provided, the function writes the result into it, and returns a reference to *out*. (See Examples)

M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972, pg. 83.

```
>>> import mars.tensor as mt
```

```
>>> mt.sinh(0).execute()
0.0
>>> mt.sinh(mt.pi*1j/2).execute()
1j
>>> mt.sinh(mt.pi*1j).execute() # (exact value is 0)
1.2246063538223773e-016j
>>> # Discrepancy due to vagaries of floating point arithmetic.
```

```
>>> # Example of providing the optional output parameter
>>> out1 = mt.zeros(1)
>>> out2 = mt.sinh([0.1], out1)
>>> out2 is out1
True
```

```
>>> # Example of ValueError due to provision of shape mis-matched `out`
>>> mt.sinh(mt.zeros((3,3)),mt.zeros((2,2))).execute()
Traceback (most recent call last):
...
ValueError:  operands could not be broadcast together with shapes (3,3) (2,2)
```

mars.tensor.cosh

mars.tensor.**cosh**(*x*, *out=None*, *where=None*, ***kwargs*)

Hyperbolic cosine, element-wise.

Equivalent to `1/2 * (mt.exp(x) + mt.exp(-x))` and `mt.cos(1j*x)`.

x [array_like] Input tensor.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

out [Tensor] Output array of same shape as *x*.

```
>>> import mars.tensor as mt
```

```
>>> mt.cosh(0).execute()
1.0
```

The hyperbolic cosine describes the shape of a hanging cable:

```
>>> import matplotlib.pyplot as plt
>>> x = mt.linspace(-4, 4, 1000)
>>> plt.plot(x.execute(), mt.cosh(x).execute())
>>> plt.show()
```

mars.tensor.tanh

mars.tensor.**tanh**(*x*, *out=None*, *where=None*, ***kwargs*)

Compute hyperbolic tangent element-wise.

Equivalent to `mt.sinh(x)/np.cosh(x)` or `-1j * mt.tan(1j*x)`.

x [array_like] Input tensor.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

y [Tensor] The corresponding hyperbolic tangent values.

If *out* is provided, the function writes the result into it, and returns a reference to *out*. (See Examples)

```
>>> import mars.tensor as mt
```

```
>>> mt.tanh((0, mt.pi*1j, mt.pi*1j/2)).execute()
array([ 0. +0.00000000e+00j,  0. -1.22460635e-16j,  0. +1.63317787e+16j])
```

```
>>> # Example of providing the optional output parameter illustrating
>>> # that what is returned is a reference to said parameter
>>> out1 = mt.zeros(1)
>>> out2 = mt.tanh([0.1], out1)
```

(continues on next page)

(continued from previous page)

```
>>> out2 is out1
True
```

```
>>> # Example of ValueError due to provision of shape mis-matched `out`
>>> mt.tanh(mt.zeros((3,3)),mt.zeros((2,2)))
Traceback (most recent call last):
...
ValueError: operands could not be broadcast together with shapes (3,3) (2,2)
```

mars.tensor.arcsinh

mars.tensor.**arcsinh**(*x*, *out=None*, *where=None*, ***kwargs*)

Inverse hyperbolic sine element-wise.

x [array_like] Input tensor.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

out [Tensor] Tensor of of the same shape as *x*.

arcsinh is a multivalued function: for each *x* there are infinitely many numbers *z* such that *sinh(z) = x*. The convention is to return the *z* whose imaginary part lies in *[-pi/2, pi/2]*.

For real-valued input data types, *arcsinh* always returns real output. For each value that cannot be expressed as a real number or infinity, it returns `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *arccos* is a complex analytical function that has branch cuts *[1j, infj]* and *[-1j, -infj]* and is continuous from the right on the former and from the left on the latter.

The inverse hyperbolic sine is also known as *asinh* or `sinh^-1`.

```
>>> import mars.tensor as mt
```

```
>>> mt.arcsinh(mt.array([mt.e, 10.0])).execute()
array([ 1.72538256,  2.99822295])
```

mars.tensor.arccosh

mars.tensor.**arccosh**(*x*, *out=None*, *where=None*, ***kwargs*)

Inverse hyperbolic cosine, element-wise.

x [array_like] Input tensor.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

arccosh [Tensor] Array of the same shape as *x*.

cosh, arcsinh, sinh, arctanh, tanh

arccosh is a multivalued function: for each *x* there are infinitely many numbers *z* such that *cosh(z) = x*. The convention is to return the *z* whose imaginary part lies in *[-pi, pi]* and the real part in `[0, inf]`.

For real-valued input data types, *arccosh* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *arccosh* is a complex analytical function that has a branch cut *[-inf, 1]* and is continuous from above on it.

```
>>> import mars.tensor as mt
```

```
>>> mt.arccosh([mt.e, 10.0]).execute()
array([ 1.65745445,  2.99322285])
>>> mt.arccosh(1).execute()
0.0
```

mars.tensor.arctanh

mars.tensor.**arctanh**(*x*, *out=None*, *where=None*, ***kwargs*)

Inverse hyperbolic tangent element-wise.

x [array_like] Input tensor.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

out [Tensor] Array of the same shape as *x*.

arctanh is a multivalued function: for each *x* there are infinitely many numbers *z* such that *tanh(z) = x*. The convention is to return the *z* whose imaginary part lies in *[-pi/2, pi/2]*.

For real-valued input data types, *arctanh* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *arctanh* is a complex analytical function that has branch cuts *[-1, -inf]* and *[1, inf]* and is continuous from above on the former and from below on the latter.

The inverse hyperbolic tangent is also known as *atanh* or `tanh^-1`.

```
>>> import mars.tensor as mt
```

```
>>> mt.arctanh([0, -0.5]).execute()
array([ 0.        , -0.54930614])
```

mars.tensor.deg2rad

mars.tensor.**deg2rad**(*x*, *out=None*, *where=None*, ***kwargs*)

Convert angles from degrees to radians.

x [array_like] Angles in degrees.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y [Tensor] The corresponding angle in radians.

rad2deg : Convert angles from radians to degrees. unwrap : Remove large jumps in angle by wrapping.

`deg2rad(x)` is `x * pi / 180`.

```
>>> import mars.tensor as mt
```

```
>>> mt.deg2rad(180).execute()
3.1415926535897931
```

mars.tensor.rad2deg

mars.tensor.**rad2deg**(*x*, *out=None*, *where=None*, ***kwargs*)

Convert angles from radians to degrees.

x [array_like] Angle in radians.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y [Tensor] The corresponding angle in degrees.

deg2rad : Convert angles from degrees to radians.

rad2deg(x) is `180 * x / pi`.

```
>>> import mars.tensor as mt
```

```
>>> mt.rad2deg(mt.pi/2).execute()
90.0
```

Bit-twiddling functions

mars.tensor.bitwise_and	Compute the bit-wise AND of two tensors element-wise.
mars.tensor.bitwise_or	Compute the bit-wise OR of two tensors element-wise.
mars.tensor.bitwise_xor	Compute the bit-wise XOR of two arrays element-wise.
mars.tensor.invert	Compute bit-wise inversion, or bit-wise NOT, element-wise.
mars.tensor.left_shift	Shift the bits of an integer to the left.
mars.tensor.right_shift	Shift the bits of an integer to the right.

mars.tensor.bitwise_and

mars.tensor.**bitwise_and**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Compute the bit-wise AND of two tensors element-wise.

Computes the bit-wise AND of the underlying binary representation of the integers in the input arrays. This ufunc implements the C/Python operator `&`.

x1, x2 [array_like] Only integer and boolean types are handled.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

out [array_like] Result.

logical_and bitwise_or bitwise_xor

The number 13 is represented by `00001101`. Likewise, 17 is represented by `00010001`. The bit-wise AND of 13 and 17 is therefore `000000001`, or 1:

```
>>> import mars.tensor as mt
```

```
>>> mt.bitwise_and(13, 17).execute()
1
```

```
>>> mt.bitwise_and(14, 13).execute()
12
>>> mt.bitwise_and([14,3], 13).execute()
array([12,  1])
```

```
>>> mt.bitwise_and([11,7], [4,25]).execute()
array([0, 1])
>>> mt.bitwise_and(mt.array([2,5,255]), mt.array([3,14,16])).execute()
array([ 2,  4, 16])
>>> mt.bitwise_and([True, True], [False, True]).execute()
array([False,  True])
```

mars.tensor.bitwise_or

mars.tensor.**bitwise_or**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Compute the bit-wise OR of two tensors element-wise.

Computes the bit-wise OR of the underlying binary representation of the integers in the input arrays. This ufunc implements the C/Python operator `|`.

x1, x2 [array_like] Only integer and boolean types are handled.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

out [array_like] Result.

logical_or bitwise_and bitwise_xor binary_repr :

Return the binary representation of the input number as a string.

The number 13 has the binaray representation `00001101`. Likewise, 16 is represented by `00010000`. The bit-wise OR of 13 and 16 is then `000111011`, or 29:

```
>>> import mars.tensor as mt
```

```
>>> mt.bitwise_or(13, 16).execute()
29
```

```
>>> mt.bitwise_or(32, 2).execute()
34
>>> mt.bitwise_or([33, 4], 1).execute()
array([33,  5])
>>> mt.bitwise_or([33, 4], [1, 2]).execute()
array([33,  6])
```

```
>>> mt.bitwise_or(mt.array([2, 5, 255]), mt.array([4, 4, 4])).execute()
array([  6,   5, 255])
>>> (mt.array([2, 5, 255]) | mt.array([4, 4, 4])).execute()
array([  6,   5, 255])
>>> mt.bitwise_or(mt.array([2, 5, 255, 2147483647], dtype=mt.int32),
...               mt.array([4, 4, 4, 2147483647], dtype=mt.int32)).execute()
array([         6,          5,        255, 2147483647])
>>> mt.bitwise_or([True, True], [False, True]).execute()
array([ True,  True])
```

mars.tensor.bitwise_xor

mars.tensor.**bitwise_xor**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Compute the bit-wise XOR of two arrays element-wise.

Computes the bit-wise XOR of the underlying binary representation of the integers in the input arrays. This ufunc implements the C/Python operator `^`.

x1, x2 [array_like] Only integer and boolean types are handled.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

out [array_like] Result.

logical_xor bitwise_and bitwise_or binary_repr :

Return the binary representation of the input number as a string.

The number 13 is represented by `00001101`. Likewise, 17 is represented by `00010001`. The bit-wise XOR of 13 and 17 is therefore `00011100`, or 28:

```
>>> import mars.tensor as mt
```

```
>>> mt.bitwise_xor(13, 17).execute()
28
```

```
>>> mt.bitwise_xor(31, 5).execute()
26
>>> mt.bitwise_xor([31,3], 5).execute()
array([26,  6])
```

```
>>> mt.bitwise_xor([31,3], [5,6]).execute()
array([26,  5])
>>> mt.bitwise_xor([True, True], [False, True]).execute()
array([ True, False])
```

mars.tensor.invert

mars.tensor.**invert**(*x*, *out=None*, *where=None*, ***kwargs*)

Compute bit-wise inversion, or bit-wise NOT, element-wise.

Computes the bit-wise NOT of the underlying binary representation of the integers in the input tensors. This ufunc implements the C/Python operator ~.

For signed integer inputs, the two's complement is returned. In a two's-complement system negative numbers are represented by the two's complement of the absolute value. This is the most common method of representing signed integers on computers[1]. A N-bit two's-complement system can represent every integer in the range -2^{N-1} to $+2^{N-1}-1$.

x [array_like] Only integer and boolean types are handled.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

out [array_like] Result.

bitwise_and, bitwise_or, bitwise_xor logical_not

bitwise_not is an alias for *invert*:

[1] Wikipedia, "Two's complement", http://en.wikipedia.org/wiki/Two's_complement

```
>>> import mars.tensor as mt
```

```
>>> mt.bitwise_not is mt.invert
True
```

We've seen that 13 is represented by `00001101`. The invert or bit-wise NOT of 13 is then:

```
>>> mt.invert(mt.array([13], dtype=mt.uint8)).execute()
array([242], dtype=uint8)
```

The result depends on the bit-width:

```
>>> mt.invert(mt.array([13], dtype=mt.uint16)).execute()
array([65522], dtype=uint16)
```

When using signed integer types the result is the two's complement of the result for the unsigned type:

```
>>> mt.invert(mt.array([13], dtype=mt.int8)).execute()
array([-14], dtype=int8)
```

Booleans are accepted as well:

```
>>> mt.invert(mt.array([True, False])).execute()
array([False,  True])
```

mars.tensor.left_shift

mars.tensor.**left_shift**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Shift the bits of an integer to the left.

Bits are shifted to the left by appending *x2* 0s at the right of *x1*. Since the internal representation of numbers is in binary format, this operation is equivalent to multiplying *x1* by `2**x2`.

x1 [array_like of integer type] Input values.

x2 [array_like of integer type] Number of zeros to append to *x1*. Has to be non-negative.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

out [tensor of integer type] Return *x1* with bits shifted *x2* times to the left.

right_shift : Shift the bits of an integer to the right.

```
>>> import mars.tensor as mt
```

```
>>> mt.left_shift(5, 2).execute()
20
```

```
>>> mt.left_shift(5, [1,2,3]).execute()
array([10, 20, 40])
```

mars.tensor.right_shift

mars.tensor.**right_shift**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Shift the bits of an integer to the right.

Bits are shifted to the right *x2*. Because the internal representation of numbers is in binary format, this operation is equivalent to dividing *x1* by `2**x2`.

x1 [array_like, int] Input values.

x2 [array_like, int] Number of bits to remove at the right of *x1*.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

out [Tensor, int] Return *x1* with bits shifted *x2* times to the right.

left_shift : Shift the bits of an integer to the left.

```
>>> import mars.tensor as mt
>>> mt.right_shift(10, 1).execute()
5
```

```
>>> mt.right_shift(10, [1,2,3]).execute()
array([5, 2, 1])
```

Comparison functions

mars.tensor.greater	Return the truth value of (x1 > x2) element-wise.
mars.tensor.greater_equal	Return the truth value of (x1 >= x2) element-wise.
mars.tensor.less	Return the truth value of (x1 < x2) element-wise.
mars.tensor.less_equal	Return the truth value of (x1 =< x2) element-wise.
mars.tensor.not_equal	Return (x1 != x2) element-wise.
mars.tensor.equal	Return (x1 == x2) element-wise.
mars.tensor.logical_and	Compute the truth value of x1 AND x2 element-wise.
mars.tensor.logical_or	Compute the truth value of x1 OR x2 element-wise.
mars.tensor.logical_xor	Compute the truth value of x1 XOR x2, element-wise.
mars.tensor.logical_not	Compute the truth value of NOT x element-wise.
mars.tensor.maximum	Element-wise maximum of tensor elements.
mars.tensor.minimum	Element-wise minimum of tensor elements.
mars.tensor.fmax	Element-wise maximum of array elements.
mars.tensor.fmin	Element-wise minimum of array elements.

mars.tensor.greater

mars.tensor.**greater**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Return the truth value of (x1 > x2) element-wise.

x1, x2 [array_like] Input tensors. If `x1.shape != x2.shape`, they must be broadcastable to a common

shape (which may be the shape of one or the other).

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

out [bool or Tensor of bool] Array of bools, or a single bool if *x1* and *x2* are scalars.

greater_equal, less, less_equal, equal, not_equal

```
>>> import mars.tensor as mt
```

```
>>> mt.greater([4,2],[2,2]).execute()
array([ True, False])
```

If the inputs are ndarrays, then np.greater is equivalent to '>'.

```
>>> a = mt.array([4,2])
>>> b = mt.array([2,2])
>>> (a > b).execute()
array([ True, False])
```

mars.tensor.greater_equal

mars.tensor.**greater_equal**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Return the truth value of (x1 >= x2) element-wise.

x1, x2 [array_like] Input tensors. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which may be the shape of one or the other).

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

out [bool or Tensor of bool] Array of bools, or a single bool if *x1* and *x2* are scalars.

greater, less, less_equal, equal, not_equal

```
>>> import mars.tensor as mt
```

```
>>> mt.greater_equal([4, 2, 1], [2, 2, 2]).execute()
array([ True, True, False])
```

mars.tensor.less

mars.tensor.**less**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Return the truth value of (x1 < x2) element-wise.

x1, x2 [array_like] Input tensors. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which may be the shape of one or the other).

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

out [bool or Tensor of bool] Array of bools, or a single bool if *x1* and *x2* are scalars.

greater, less_equal, greater_equal, equal, not_equal

```
>>> import mars.tensor as mt
```

```
>>> mt.less([1, 2], [2, 2]).execute()
array([ True, False])
```

mars.tensor.less_equal

mars.tensor.**less_equal**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Return the truth value of (x1 =< x2) element-wise.

x1, x2 [array_like] Input tensors. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which may be the shape of one or the other).

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

out [bool or tensor of bool] Array of bools, or a single bool if *x1* and *x2* are scalars.

greater, less, greater_equal, equal, not_equal

```
>>> import mars.tensor as mt
```

```
>>> mt.less_equal([4, 2, 1], [2, 2, 2]).execute()
array([False,  True,  True])
```

mars.tensor.not_equal

mars.tensor.**not_equal**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Return (x1 != x2) element-wise.

x1, x2 [array_like] Input tensors.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

not_equal [tensor bool, scalar bool] For each element in *x1, x2*, return True if *x1* is not equal to *x2* and False otherwise.

equal, greater, greater_equal, less, less_equal

```
>>> import mars.tensor as mt
```

```
>>> mt.not_equal([1.,2.], [1., 3.]).execute()
array([False,  True])
>>> mt.not_equal([1, 2], [[1, 3],[1, 4]]).execute()
array([[False,  True],
       [False,  True]])
```

mars.tensor.equal

mars.tensor.**equal**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Return (x1 == x2) element-wise.

x1, x2 [array_like] Input tensors of the same shape.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs** For other keyword-only arguments, see the ufunc docs.

out [Tensor or bool] Output tensor of bools, or a single bool if x1 and x2 are scalars.

not_equal, greater_equal, less_equal, greater, less

```
>>> import mars.tensor as mt
```

```
>>> mt.equal([0, 1, 3], mt.arange(3)).execute()
array([ True,  True, False])
```

What is compared are values, not types. So an int (1) and a tensor of length one can evaluate as True:

```
>>> mt.equal(1, mt.ones(1))
array([ True])
```

mars.tensor.logical_and

mars.tensor.**logical_and**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Compute the truth value of x1 AND x2 element-wise.

x1, x2 [array_like] Input tensors. *x1* and *x2* must be of the same shape.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

y [Tensor or bool] Boolean result with the same shape as *x1* and *x2* of the logical AND operation on corresponding elements of *x1* and *x2*.

logical_or, logical_not, logical_xor bitwise_and

```
>>> import mars.tensor as mt
```

```
>>> mt.logical_and(True, False).execute()
False
>>> mt.logical_and([True, False], [False, False]).execute()
array([False, False])
```

```
>>> x = mt.arange(5)
>>> mt.logical_and(x>1, x<4).execute()
array([False, False,  True,  True, False])
```

mars.tensor.logical_or

mars.tensor.**logical_or**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Compute the truth value of x1 OR x2 element-wise.

x1, x2 [array_like] Logical OR is applied to the elements of *x1* and *x2*. They have to be of the same shape.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

y [Tensor or bool] Boolean result with the same shape as *x1* and *x2* of the logical OR operation on elements of *x1* and *x2*.

logical_and, logical_not, logical_xor bitwise_or

```
>>> import mars.tensor as mt
```

```
>>> mt.logical_or(True, False).execute()
True
>>> mt.logical_or([True, False], [False, False]).execute()
array([ True, False])
```

```
>>> x = mt.arange(5)
>>> mt.logical_or(x < 1, x > 3).execute()
array([ True, False, False, False,  True])
```

mars.tensor.logical_xor

mars.tensor.**logical_xor**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Compute the truth value of x1 XOR x2, element-wise.

x1, x2 [array_like] Logical XOR is applied to the elements of *x1* and *x2*. They must be broadcastable to the same shape.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y [bool or Tensor of bool] Boolean result of the logical XOR operation applied to the elements of *x1* and *x2*; the shape is determined by whether or not broadcasting of one or both arrays was required.

logical_and, logical_or, logical_not, bitwise_xor

```
>>> import mars.tensor as mt
```

```
>>> mt.logical_xor(True, False).execute()
True
>>> mt.logical_xor([True, True, False, False], [True, False, True, False]).
↪execute()
array([False,  True,  True, False])
```

```
>>> x = mt.arange(5)
>>> mt.logical_xor(x < 1, x > 3).execute()
array([ True, False, False, False,  True])
```

Simple example showing support of broadcasting

```
>>> mt.logical_xor(0, mt.eye(2)).execute()
array([[ True, False],
       [False,  True]])
```

mars.tensor.logical_not

mars.tensor.**logical_not**(*x*, *out=None*, *where=None*, ***kwargs*)

Compute the truth value of NOT x element-wise.

x [array_like] Logical NOT is applied to the elements of *x*.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y [bool or Tensor of bool] Boolean result with the same shape as *x* of the NOT operation on elements of *x*.

logical_and, logical_or, logical_xor

```
>>> import mars.tensor as mt
```

```
>>> mt.logical_not(3).execute()
False
>>> mt.logical_not([True, False, 0, 1]).execute()
array([False,  True,  True, False])
```

```
>>> x = mt.arange(5)
>>> mt.logical_not(x<3).execute()
array([False, False, False,  True,  True])
```

mars.tensor.maximum

mars.tensor.**maximum**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Element-wise maximum of tensor elements.

Compare two tensors and returns a new array containing the element-wise maxima. If one of the elements being compared is a NaN, then that element is returned. If both elements are NaNs then the first is returned. The latter distinction is important for complex NaNs, which are defined as at least one of the real or imaginary parts being a NaN. The net effect is that NaNs are propagated.

x1, x2 [array_like] The tensors holding the elements to be compared. They must have the same shape, or shapes that can be broadcast to a single shape.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y [ndarray or scalar] The maximum of *x1* and *x2*, element-wise. Returns scalar if both *x1* and *x2* are scalars.

minimum : Element-wise minimum of two tensors, propagates NaNs.

fmax : Element-wise maximum of two tensors, ignores NaNs.

amax : The maximum value of a tensor along a given axis, propagates NaNs.

nanmax : The maximum value of a tensor along a given axis, ignores NaNs.

fmin, amin, nanmin

The maximum is equivalent to `mt.where(x1 >= x2, x1, x2)` when neither x1 nor x2 are nans, but it is faster and does proper broadcasting.

```
>>> import mars.tensor as mt
```

```
>>> mt.maximum([2, 3, 4], [1, 5, 2]).execute()
array([2, 5, 4])
```

```
>>> mt.maximum(mt.eye(2), [0.5, 2]).execute() # broadcasting
array([[ 1. ,  2. ],
       [ 0.5,  2. ]])
```

```
>>> mt.maximum([mt.nan, 0, mt.nan], [0, mt.nan, mt.nan]).execute()
array([ NaN,  NaN,  NaN])
>>> mt.maximum(mt.Inf, 1).execute()
inf
```

mars.tensor.minimum

mars.tensor.**minimum**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Element-wise minimum of tensor elements.

Compare two tensors and returns a new tensor containing the element-wise minima. If one of the elements being compared is a NaN, then that element is returned. If both elements are NaNs then the first is returned. The latter distinction is important for complex NaNs, which are defined as at least one of the real or imaginary parts being a NaN. The net effect is that NaNs are propagated.

x1, x2 [array_like] The tensors holding the elements to be compared. They must have the same shape, or shapes that can be broadcast to a single shape.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y [Tensor or scalar] The minimum of *x1* and *x2*, element-wise. Returns scalar if both *x1* and *x2* are scalars.

maximum : Element-wise maximum of two tensors, propagates NaNs.

fmin : Element-wise minimum of two tensors, ignores NaNs.

amin : The minimum value of a tensor along a given axis, propagates NaNs.

nanmin : The minimum value of a tenosr along a given axis, ignores NaNs.

fmax, amax, nanmax

The minimum is equivalent to `mt.where(x1 <= x2, x1, x2)` when neither x1 nor x2 are NaNs, but it is faster and does proper broadcasting.

```
>>> import mars.tensor as mt
```

```
>>> mt.minimum([2, 3, 4], [1, 5, 2]).execute()
array([1, 3, 2])
```

```
>>> mt.minimum(mt.eye(2), [0.5, 2]).execute() # broadcasting
array([[ 0.5,  0. ],
       [ 0. ,  1. ]])
```

```
>>> mt.minimum([mt.nan, 0, mt.nan],[0, mt.nan, mt.nan]).execute()
array([ NaN,  NaN,  NaN])
>>> mt.minimum(-mt.Inf, 1).execute()
-inf
```

mars.tensor.fmax

mars.tensor.**fmax**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Element-wise maximum of array elements.

Compare two tensors and returns a new tensor containing the element-wise maxima. If one of the elements being compared is a NaN, then the non-nan element is returned. If both elements are NaNs then the first is returned. The latter distinction is important for complex NaNs, which are defined as at least one of the real or imaginary parts being a NaN. The net effect is that NaNs are ignored when possible.

x1, x2 [array_like] The tensors holding the elements to be compared. They must have the same shape.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y [Tensor or scalar] The maximum of *x1* and *x2*, element-wise. Returns scalar if both *x1* and *x2* are scalars.

fmin : Element-wise minimum of two tensors, ignores NaNs.

maximum : Element-wise maximum of two tensors, propagates NaNs.

amax : The maximum value of an tensor along a given axis, propagates NaNs.

nanmax : The maximum value of an tensor along a given axis, ignores NaNs.

minimum, amin, nanmin

The fmax is equivalent to `mt.where(x1 >= x2, x1, x2)` when neither x1 nor x2 are NaNs, but it is faster and does proper broadcasting.

```
>>> import mars.tensor as mt
```

```
>>> mt.fmax([2, 3, 4], [1, 5, 2]).execute()
array([ 2.,  5.,  4.])
```

```
>>> mt.fmax(mt.eye(2), [0.5, 2]).execute()
array([[ 1. ,  2. ],
       [ 0.5,  2. ]])
```

```
>>> mt.fmax([mt.nan, 0, mt.nan],[0, mt.nan, mt.nan]).execute()
array([  0.,   0.,  NaN])
```

mars.tensor.fmin

mars.tensor.**fmin**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Element-wise minimum of array elements.

Compare two tensors and returns a new tensor containing the element-wise minima. If one of the elements being compared is a NaN, then the non-nan element is returned. If both elements are NaNs then the first is returned. The latter distinction is important for complex NaNs, which are defined as at least one of the real or imaginary parts being a NaN. The net effect is that NaNs are ignored when possible.

x1, x2 [array_like] The tensors holding the elements to be compared. They must have the same shape.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

y [Tensor or scalar] The minimum of *x1* and *x2*, element-wise. Returns scalar if both *x1* and *x2* are scalars.

fmax : Element-wise maximum of two tensors, ignores NaNs.

minimum : Element-wise minimum of two tensors, propagates NaNs.

amin : The minimum value of a tensor along a given axis, propagates NaNs.

nanmin : The minimum value of a tensor along a given axis, ignores NaNs.

maximum, amax, nanmax

The fmin is equivalent to `mt.where(x1 <= x2, x1, x2)` when neither x1 nor x2 are NaNs, but it is faster and does proper broadcasting.

```
>>> import mars.tensor as mt
```

```
>>> mt.fmin([2, 3, 4], [1, 5, 2]).execute()
array([1, 3, 2])
```

```
>>> mt.fmin(mt.eye(2), [0.5, 2]).execute()
array([[ 0.5,  0. ],
       [ 0. ,  1. ]])
```

```
>>> mt.fmin([mt.nan, 0, mt.nan],[0, mt.nan, mt.nan]).execute()
array([  0.,   0.,  NaN])
```

Floating point values

mars.tensor.isfinite	Test element-wise for finiteness (not infinity or not Not a Number).
mars.tensor.isinf	Test element-wise for positive or negative infinity.
mars.tensor.isnan	Test element-wise for NaN and return result as a boolean tensor.
mars.tensor.signbit	Returns element-wise True where signbit is set (less than zero).
mars.tensor.copysign	Change the sign of x1 to that of x2, element-wise.
mars.tensor.nextafter	Return the next floating-point value after x1 towards x2, element-wise.
mars.tensor.modf	Return the fractional and integral parts of a tensor, element-wise.
mars.tensor.ldexp	Returns x1 * 2**x2, element-wise.
mars.tensor.frexp	Decompose the elements of x into mantissa and twos exponent.

Continued on next page

Table 6 – continued from previous page

mars.tensor.fmod	Return the element-wise remainder of division.
mars.tensor.floor	Return the floor of the input, element-wise.
mars.tensor.ceil	Return the ceiling of the input, element-wise.
mars.tensor.trunc	Return the truncated value of the input, element-wise.

mars.tensor.isfinite

mars.tensor.**isfinite**(*x*, *out=None*, *where=None*, ***kwargs*)

Test element-wise for finiteness (not infinity or not Not a Number).

The result is returned as a boolean tensor.

x [array_like] Input values.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y [Tensor, bool] For scalar input, the result is a new boolean with value True if the input is finite; otherwise the value is False (input is either positive infinity, negative infinity or Not a Number).

For array input, the result is a boolean array with the same dimensions as the input and the values are True if the corresponding element of the input is finite; otherwise the values are False (element is either positive infinity, negative infinity or Not a Number).

isinf, isneginf, isposinf, isnan

Not a Number, positive infinity and negative infinity are considered to be non-finite.

Mars uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity. Also that positive infinity is not equivalent to negative infinity. But infinity is equivalent to positive infinity. Errors result if the second argument is also supplied when *x* is a scalar input, or if first and second arguments have different shapes.

```
>>> import mars.tensor as mt
```

```
>>> mt.isfinite(1).execute()
True
>>> mt.isfinite(0).execute()
True
>>> mt.isfinite(mt.nan).execute()
False
>>> mt.isfinite(mt.inf).execute()
False
>>> mt.isfinite(mt.NINF).execute()
False
>>> mt.isfinite([mt.log(-1.).execute(),1.,mt.log(0).execute()]).execute()
array([False,  True, False])
```

```
>>> x = mt.array([-mt.inf, 0., mt.inf])
>>> y = mt.array([2, 2, 2])
>>> mt.isfinite(x, y).execute()
```

(continues on next page)

(continued from previous page)

```
array([0, 1, 0])
>>> y.execute()
array([0, 1, 0])
```

mars.tensor.isinf

mars.tensor.**isinf**(*x*, *out=None*, *where=None*, ***kwargs*)

Test element-wise for positive or negative infinity.

Returns a boolean array of the same shape as *x*, True where `x == +/-inf`, otherwise False.

x [array_like] Input values

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y [bool (scalar) or boolean Tensor] For scalar input, the result is a new boolean with value True if the input is positive or negative infinity; otherwise the value is False.

For tensor input, the result is a boolean tensor with the same shape as the input and the values are True where the corresponding element of the input is positive or negative infinity; elsewhere the values are False. If a second argument was supplied the result is stored there. If the type of that array is a numeric type the result is represented as zeros and ones, if the type is boolean then as False and True, respectively. The return value *y* is then a reference to that tensor.

isneginf, isposinf, isnan, isfinite

Mars uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754).

Errors result if the second argument is supplied when the first argument is a scalar, or if the first and second arguments have different shapes.

```
>>> import mars.tensor as mt
```

```
>>> mt.isinf(mt.inf).execute()
True
>>> mt.isinf(mt.nan).execute()
False
>>> mt.isinf(mt.NINF).execute()
True
>>> mt.isinf([mt.inf, -mt.inf, 1.0, mt.nan]).execute()
array([ True,  True, False, False])
```

```
>>> x = mt.array([-mt.inf, 0., mt.inf])
>>> y = mt.array([2, 2, 2])
>>> mt.isinf(x, y).execute()
array([1, 0, 1])
>>> y.execute()
array([1, 0, 1])
```

mars.tensor.isnan

mars.tensor.**isnan**(*x*, *out=None*, *where=None*, ***kwargs*)

Test element-wise for NaN and return result as a boolean tensor.

x [array_like] Input tensor.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y [Tensor or bool] For scalar input, the result is a new boolean with value True if the input is NaN; otherwise the value is False.

For array input, the result is a boolean tensor of the same dimensions as the input and the values are True if the corresponding element of the input is NaN; otherwise the values are False.

isinf, isneginf, isposinf, isfinite, isnat

Mars uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity.

```
>>> import mars.tensor as mt
```

```
>>> mt.isnan(mt.nan).execute()
True
>>> mt.isnan(mt.inf).execute()
False
>>> mt.isnan([mt.log(-1.).execute(),1.,mt.log(0).execute()]).execute()
array([ True, False, False])
```

mars.tensor.signbit

mars.tensor.**signbit**(*x*, *out=None*, *where=None*, ***kwargs*)

Returns element-wise True where signbit is set (less than zero).

x [array_like] The input value(s).

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

result [Tensor of bool] Output tensor, or reference to *out* if that was supplied.

```
>>> import mars.tensor as mt
```

```
>>> mt.signbit(-1.2).execute()
True
```

(continues on next page)

(continued from previous page)

```
>>> mt.signbit(mt.array([1, -2.3, 2.1])).execute()
array([False,  True, False])
```

mars.tensor.copysign

mars.tensor.**copysign**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Change the sign of x1 to that of x2, element-wise.

If both arguments are arrays or sequences, they have to be of the same length. If *x2* is a scalar, its sign will be copied to all elements of *x1*.

x1 [array_like] Values to change the sign of.

x2 [array_like] The sign of *x2* is copied to *x1*.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

out [array_like] The values of *x1* with the sign of *x2*.

```
>>> import mars.tensor as mt
```

```
>>> mt.copysign(1.3, -1).execute()
-1.3
>>> (1/mt.copysign(0, 1)).execute()
inf
>>> (1/mt.copysign(0, -1)).execute()
-inf
```

```
>>> mt.copysign([-1, 0, 1], -1.1).execute()
array([-1., -0., -1.])
>>> mt.copysign([-1, 0, 1], mt.arange(3)-1).execute()
array([-1.,  0.,  1.])
```

mars.tensor.nextafter

mars.tensor.**nextafter**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Return the next floating-point value after x1 towards x2, element-wise.

x1 [array_like] Values to find the next representable value of.

x2 [array_like] The direction where to look for the next representable value of *x1*.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

out [array_like] The next representable values of *x1* in the direction of *x2*.

```
>>> import mars.tensor as mt
```

```
>>> eps = mt.finfo(mt.float64).eps
>>> (mt.nextafter(1, 2) == eps + 1).execute()
True
>>> (mt.nextafter([1, 2], [2, 1]) == [eps + 1, 2 - eps]).execute()
array([ True,  True])
```

mars.tensor.modf

mars.tensor.**modf**(*x*, *out1=None*, *out2=None*, *out=None*, *where=None*, ***kwargs*)

Return the fractional and integral parts of a tensor, element-wise.

The fractional and integral parts are negative if the given number is negative.

x [array_like] Input tensor.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y1 [Tensor] Fractional part of *x*.

y2 [Tensor] Integral part of *x*.

For integer input the return values are floats.

divmod [`divmod(x, 1)` is equivalent to `modf` with the return values] switched, except it always has a positive remainder.

```
>>> import mars.tensor as mt
>>> from mars.session import new_session
```

```
>>> sess = new_session().as_default()
>>> sess.run(mt.modf([0, 3.5]))
(array([ 0. ,  0.5]), array([ 0.,  3.]))
>>> sess.run(mt.modf(-0.5))
(-0.5, -0)
```

mars.tensor.ldexp

mars.tensor.**ldexp**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

Returns x1 * 2**x2, element-wise.

The mantissas *x1* and twos exponents *x2* are used to construct floating point numbers `x1 * 2**x2`.

x1 [array_like] Tensor of multipliers.

x2 [array_like, int] Tensor of twos exponents.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

y [Tensor or scalar] The result of `x1 * 2**x2`.

frexp : Return (y1, y2) from `x = y1 * 2**y2`, inverse to *ldexp*.

Complex dtypes are not supported, they will raise a TypeError.

ldexp is useful as the inverse of *frexp*, if used by itself it is more clear to simply use the expression `x1 * 2**x2`.

```
>>> import mars.tensor as mt
```

```
>>> mt.ldexp(5, mt.arange(4)).execute()
array([  5.,  10.,  20.,  40.], dtype=float32)
```

```
>>> x = mt.arange(6)
>>> mt.ldexp(*mt.frexp(x)).execute()
array([ 0.,  1.,  2.,  3.,  4.,  5.])
```

mars.tensor.frexp

mars.tensor.**frexp**(*x*, *out1=None*, *out2=None*, *out=None*, *where=None*, ***kwargs*)

Decompose the elements of x into mantissa and twos exponent.

Returns (*mantissa*, *exponent*), where *x = mantissa * 2**exponent‘*. The mantissa is lies in the open interval(-1, 1), while the twos exponent is a signed integer.

x [array_like] Tensor of numbers to be decomposed.

out1 [Tensor, optional] Output tensor for the mantissa. Must have the same shape as *x*.

out2 [Tensor, optional] Output tensor for the exponent. Must have the same shape as *x*.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

(mantissa, exponent) [tuple of tensors, (float, int)] *mantissa* is a float array with values between -1 and 1. *exponent* is an int array which represents the exponent of 2.

ldexp : Compute `y = x1 * 2**x2`, the inverse of *frexp*.

Complex dtypes are not supported, they will raise a TypeError.

```
>>> import mars.tensor as mt
>>> from mars.session import new_session
```

```
>>> x = mt.arange(9)
>>> y1, y2 = mt.frexp(x)
```

```
>>> sess = new_session().as_default()
>>> y1_result, y2_result = sess.run(y1, y2)
>>> y1_result
array([ 0.   ,  0.5  ,  0.5  ,  0.75 ,  0.5  ,  0.625,  0.75 ,  0.875,
        0.5  ])
>>> y2_result
array([0, 1, 2, 2, 3, 3, 3, 3, 4])
>>> (y1 * 2**y2).execute(session=sess)
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.])
```

mars.tensor.floor

mars.tensor.**floor**(*x*, *out=None*, *where=None*, ***kwargs*)

Return the floor of the input, element-wise.

The floor of the scalar *x* is the largest integer *i*, such that *i* <= *x*. It is often denoted as $\lfloor x \rfloor$.

x [array_like] Input data.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y [Tensor or scalar] The floor of each element in *x*.

ceil, trunc, rint

Some spreadsheet programs calculate the "floor-towards-zero", in other words `floor(-2.5) == -2`. NumPy instead uses the definition of *floor* where *floor(-2.5) == -3*.

```
>>> import mars.tensor as mt
```

```
>>> a = mt.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> mt.floor(a).execute()
array([-2., -2., -1.,  0.,  1.,  1.,  2.])
```

mars.tensor.ceil

mars.tensor.**ceil**(*x*, *out=None*, *where=None*, ***kwargs*)

Return the ceiling of the input, element-wise.

The ceil of the scalar *x* is the smallest integer *i*, such that *i* >= *x*. It is often denoted as $\lceil x \rceil$.

x [array_like] Input data.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

y [Tensor or scalar] The ceiling of each element in *x*, with *float* dtype.

floor, trunc, rint

```
>>> import mars.tensor as mt
```

```
>>> a = mt.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> mt.ceil(a).execute()
array([-1., -1., -0.,  1.,  2.,  2.,  2.])
```

mars.tensor.trunc

mars.tensor.**trunc**(*x*, *out=None*, *where=None*, ***kwargs*)

Return the truncated value of the input, element-wise.

The truncated value of the scalar *x* is the nearest integer *i* which is closer to zero than *x* is. In short, the fractional part of the signed number *x* is discarded.

x [array_like] Input data.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

y [Tensor or scalar] The truncated value of each element in *x*.

ceil, floor, rint

```
>>> import mars.tensor as mt
```

```
>>> a = mt.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> mt.trunc(a).execute()
array([-1., -1., -0.,  0.,  1.,  1.,  2.])
```

2.6 Routines

The following pages describe Numpy-compatible routines. These functions cover a subset of NumPy routines.

2.6.1 Tensor Creation Routines

Basic creation routines

mars.tensor.empty	Return a new tensor of given shape and type, without initializing entries.
mars.tensor.empty_like	Return a new tensor with the same shape and type as a given tensor.
mars.tensor.eye	Return a 2-D tensor with ones on the diagonal and zeros elsewhere.
mars.tensor.identity	Return the identity tensor.
mars.tensor.ones	Return a new tensor of given shape and type, filled with ones.
mars.tensor.ones_like	Return a tensor of ones with the same shape and type as a given tensor.
mars.tensor.zeros	Return a new tensor of given shape and type, filled with zeros.
mars.tensor.zeros_like	Return a tensor of zeros with the same shape and type as a given tensor.
mars.tensor.full	Return a new tensor of given shape and type, filled with *fill_value*.

mars.tensor.empty

mars.tensor.**empty**(*shape*, *dtype=None*, *chunk_size=None*, *gpu=False*, *order='C'*)

Return a new tensor of given shape and type, without initializing entries. Parameters ———- shape : int or tuple of int

Shape of the empty tensor

dtype [data-type, optional] Desired output data-type.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

order [{'C', 'F'}, optional, default: 'C'] Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.

out [Tensor] Tensor of uninitialized (arbitrary) data of the given shape, dtype, and order. Object arrays will be initialized to None.

empty_like, zeros, ones Notes —– *empty*, unlike *zeros*, does not set the array values to zero, and may therefore be marginally faster. On the other hand, it requires the user to manually set all the values in the array, and should be used with caution. Examples ——— >>> import mars.tensor as mt >>> mt.empty([2, 2]).execute() array([[-9.74499359e+001, 6.69583040e-309],

[2.13182611e-314, 3.06959433e-309]]) #random

```
>>> mt.empty([2, 2], dtype=int).execute()
array([[-1073741821, -1067949133],
       [  496041986,    19249760]])                     #random
```

mars.tensor.empty_like

mars.tensor.**empty_like**(*a*, *dtype=None*, *gpu=None*, *order='K'*)

Return a new tensor with the same shape and type as a given tensor. Parameters ———- a : array_like

The shape and data-type of *a* define these same attributes of the returned tensor.

dtype [data-type, optional] Overrides the data type of the result.

gpu [bool, optional] Allocate the tensor on GPU if True, None as default

order [{'C', 'F', 'A', or 'K'}, optional] Overrides the memory layout of the result. 'C' means C-order, 'F' means F-order, 'A' means 'F' if `prototype` is Fortran contiguous, 'C' otherwise. 'K' means match the layout of `prototype` as closely as possible.

out [Tensor] Array of uninitialized (arbitrary) data with the same shape and type as *a*.

ones_like : Return a tensor of ones with shape and type of input. zeros_like : Return a tensor of zeros with shape and type of input. empty : Return a new uninitialized tensor. ones : Return a new tensor setting values to one. zeros : Return a new tensor setting values to zero. Notes —– This function does *not* initialize the returned tensor; to do that use *zeros_like* or *ones_like* instead. It may be marginally faster than the functions that do set the array values. Examples ——– >>> import mars.tensor as mt >>> a = ([1,2,3], [4,5,6]) # a is array-like >>> mt.empty_like(a).execute() array([[-1073741821, -1073741821, 3], #ranm

[0, 0, -1073741821]])

```
>>> a = mt.array([[1., 2., 3.],[4.,5.,6.]])
>>> mt.empty_like(a).execute()
array([[ -2.00000715e+000,   1.48219694e-323,  -2.00000572e+000],#random
       [  4.38791518e-305,  -2.00000715e+000,   4.17269252e-309]])
```

mars.tensor.eye

mars.tensor.**eye**(*N*, *M=None*, *k=0*, *dtype=None*, *sparse=False*, *gpu=False*, *chunk_size=None*, *order='C'*)

Return a 2-D tensor with ones on the diagonal and zeros elsewhere.

N [int] Number of rows in the output.

M [int, optional] Number of columns in the output. If None, defaults to *N*.

k [int, optional] Index of the diagonal: 0 (the default) refers to the main diagonal, a positive value refers to an upper diagonal, and a negative value to a lower diagonal.

dtype [data-type, optional] Data-type of the returned tensor.

sparse: bool, optional Create sparse tensor if True, False as default

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

order [{'C', 'F'}, optional] Whether the output should be stored in row-major (C-style) or column-major (Fortran-style) order in memory.

I [Tensor of shape (N,M)] An tensor where all elements are equal to zero, except for the *k*-th diagonal, whose values are equal to one.

identity : (almost) equivalent function diag : diagonal 2-D tensor from a 1-D tensor specified by the user.

```
>>> import mars.tensor as mt
```

```
>>> mt.eye(2, dtype=int).execute()
array([[1, 0],
       [0, 1]])
>>> mt.eye(3, k=1).execute()
array([[ 0.,  1.,  0.],
       [ 0.,  0.,  1.],
       [ 0.,  0.,  0.]])
```

mars.tensor.identity

mars.tensor.**identity**(*n*, *dtype=None*, *sparse=False*, *gpu=False*, *chunk_size=None*)

Return the identity tensor.

The identity tensor is a square array with ones on the main diagonal.

n [int] Number of rows (and columns) in *n* x *n* output.

dtype [data-type, optional] Data-type of the output. Defaults to `float`.

sparse: bool, optional Create sparse tensor if True, False as default

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

chunks [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

out [Tensor] *n* x *n* array with its main diagonal set to one, and all other elements 0.

```
>>> import mars.tensor as mt
```

```
>>> mt.identity(3).execute()
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

mars.tensor.ones

mars.tensor.**ones**(*shape*, *dtype=None*, *chunk_size=None*, *gpu=False*, *order='C'*)

Return a new tensor of given shape and type, filled with ones.

shape [int or sequence of ints] Shape of the new tensor, e.g., `(2, 3)` or `2`.

dtype [data-type, optional] The desired data-type for the tensor, e.g., *mt.int8*. Default is *mt.float64*.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

order [{'C', 'F'}, optional, default: C] Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.

out [Tensor] Tensor of ones with the given shape, dtype, and order.

zeros, ones_like

```
>>> import mars.tensor as mt
```

```
>>> mt.ones(5).execute()
array([ 1.,  1.,  1.,  1.,  1.])
```

```
>>> mt.ones((5,), dtype=int).execute()
array([1, 1, 1, 1, 1])
```

```
>>> mt.ones((2, 1)).execute()
array([[ 1.],
       [ 1.]])
```

```
>>> s = (2,2)
>>> mt.ones(s).execute()
array([[ 1.,  1.],
       [ 1.,  1.]])
```

mars.tensor.ones_like

mars.tensor.**ones_like**(*a*, *dtype=None*, *gpu=None*, *order='K'*)

Return a tensor of ones with the same shape and type as a given tensor.

a [array_like] The shape and data-type of *a* define these same attributes of the returned tensor.

dtype [data-type, optional] Overrides the data type of the result.

gpu [bool, optional] Allocate the tensor on GPU if True, None as default

order [{'C', 'F', 'A', or 'K'}, optional] Overrides the memory layout of the result. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible.

out [Tensor] Tensor of ones with the same shape and type as *a*.

zeros_like : Return a tensor of zeros with shape and type of input. empty_like : Return a empty tensor with shape and type of input. zeros : Return a new tensor setting values to zero. ones : Return a new tensor setting values to one. empty : Return a new uninitialized tensor.

```
>>> import mars.tensor as mt
```

```
>>> x = mt.arange(6)
>>> x = x.reshape((2, 3))
>>> x.execute()
array([[0, 1, 2],
       [3, 4, 5]])
>>> mt.ones_like(x).execute()
array([[1, 1, 1],
       [1, 1, 1]])
```

```
>>> y = mt.arange(3, dtype=float)
>>> y.execute()
array([ 0.,  1.,  2.])
>>> mt.ones_like(y).execute()
array([ 1.,  1.,  1.])
```

mars.tensor.zeros

mars.tensor.**zeros**(*shape*, *dtype=None*, *chunk_size=None*, *gpu=False*, *sparse=False*, *order='C'*)

Return a new tensor of given shape and type, filled with zeros. Parameters ———- shape : int or sequence of ints

Shape of the new tensor, e.g., `(2, 3)` or `2`.

dtype [data-type, optional] The desired data-type for the array, e.g., *mt.int8*. Default is *mt.float64*.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

sparse: bool, optional Create sparse tensor if True, False as default

order [{'C', 'F'}, optional, default: 'C'] Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.

out [Tensor] Tensor of zeros with the given shape, dtype, and order.

zeros_like : Return a tensor of zeros with shape and type of input. ones_like : Return a tensor of ones with shape and type of input. empty_like : Return a empty tensor with shape and type of input. ones : Return a new tensor setting values to one. empty : Return a new uninitialized tensor. Examples ——— >>> import mars.tensor as mt >>> mt.zeros(5).execute() array([0., 0., 0., 0., 0.]) >>> mt.zeros((5,), dtype=int).execute() array([0, 0, 0, 0, 0]) >>> mt.zeros((2, 1)).execute() array([[0.],

[0.]])

```
>>> s = (2,2)
>>> mt.zeros(s).execute()
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> mt.zeros((2,), dtype=[('x', 'i4'), ('y', 'i4')]).execute() # custom dtype
array([(0, 0), (0, 0)],
      dtype=[('x', '<i4'), ('y', '<i4')])
```

mars.tensor.zeros_like

mars.tensor.**zeros_like**(*a*, *dtype=None*, *gpu=None*, *order='K'*)

Return a tensor of zeros with the same shape and type as a given tensor. Parameters ———- a : array_like

The shape and data-type of *a* define these same attributes of the returned array.

dtype [data-type, optional] Overrides the data type of the result.

gpu [bool, optional] Allocate the tensor on GPU if True, None as default

order [{'C', 'F', 'A', or 'K'}, optional] Overrides the memory layout of the result. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible.

out [Tensor] tensor of zeros with the same shape and type as *a*.

ones_like : Return an array of ones with shape and type of input. empty_like : Return an empty array with shape and type of input. zeros : Return a new array setting values to zero. ones : Return a new array setting

values to one. empty : Return a new uninitialized array. Examples ——— >>> import mars.tensr as mt >>> x = mt.arange(6) >>> x = x.reshape((2, 3)) >>> x.execute() array([[0, 1, 2],

[3, 4, 5]])

```
>>> mt.zeros_like(x).execute()
array([[0, 0, 0],
       [0, 0, 0]])
>>> y = mt.arange(3, dtype=float)
>>> y.execute()
array([ 0.,  1.,  2.])
>>> mt.zeros_like(y).execute()
array([ 0.,  0.,  0.])
```

mars.tensor.full

mars.tensor.**full**(*shape*, *fill_value*, *dtype=None*, *chunk_size=None*, *gpu=False*, *order='C'*)

Return a new tensor of given shape and type, filled with *fill_value*.

shape [int or sequence of ints] Shape of the new tensor, e.g., `(2, 3)` or `2`.

fill_value [scalar] Fill value.

dtype [data-type, optional]

The desired data-type for the tensor The default, *None*, means *np.array(fill_value).dtype*.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

order [{'C', 'F'}, optional] Whether to store multidimensional data in C- or Fortran-contiguous (row- or column-wise) order in memory.

out [Tensor] Tensor of *fill_value* with the given shape, dtype, and order.

zeros_like : Return a tensor of zeros with shape and type of input. ones_like : Return a tensor of ones with shape and type of input. empty_like : Return an empty tensor with shape and type of input. full_like : Fill a tensor with shape and type of input. zeros : Return a new tensor setting values to zero. ones : Return a new tensor setting values to one. empty : Return a new uninitialized tensor.

```
>>> import mars.tensor as mt
```

```
>>> mt.full((2, 2), mt.inf).execute()
array([[ inf,  inf],
       [ inf,  inf]])
>>> mt.full((2, 2), 10).execute()
array([[10, 10],
       [10, 10]])
```

Creation from other data

mars.tensor.array	Create a tensor.
mars.tensor.asarray	Convert the input to an array.

mars.tensor.asarray

mars.tensor.**asarray**(*x*, *dtype=None*, *order=None*, *chunk_size=None*)

Convert the input to an array.

a [array_like] Input data, in any form that can be converted to a tensor. This includes lists, lists of tuples, tuples, tuples of tuples, tuples of lists and tensors.

dtype [data-type, optional] By default, the data-type is inferred from the input data.

order [{'C', 'F'}, optional] Whether to use row-major (C-style) or column-major (Fortran-style) memory representation.

chunk_size: int, tuple, optional Specifies chunk size for each dimension.

out [Tensor] Tensor interpretation of *a*. No copy is performed if the input is already an ndarray with matching dtype and order. If *a* is a subclass of ndarray, a base class ndarray is returned.

ascontiguousarray : Convert input to a contiguous tensor. asfortranarray : Convert input to a tensor with column-major

memory order.

Convert a list into a tensor:

```
>>> import mars.tensor as mt
```

```
>>> a = [1, 2]
>>> mt.asarray(a).execute()
array([1, 2])
```

Existing arrays are not copied:

```
>>> a = mt.array([1, 2])
>>> mt.asarray(a) is a
True
```

If *dtype* is set, array is copied only if dtype does not match:

```
>>> a = mt.array([1, 2], dtype=mt.float32)
>>> mt.asarray(a, dtype=mt.float32) is a
True
>>> mt.asarray(a, dtype=mt.float64) is a
False
```

Numerical ranges

mars.tensor.arange	Return evenly spaced values within a given interval.
mars.tensor.linspace	Return evenly spaced numbers over a specified interval.
mars.tensor.meshgrid	Return coordinate matrices from coordinate vectors.
mars.tensor.mgrid	Construct a multi-dimensional "meshgrid".
mars.tensor.ogrid	Construct a multi-dimensional "meshgrid".

mars.tensor.arange

mars.tensor.**arange**(**args*, ***kwargs*)

Return evenly spaced values within a given interval.

Values are generated within the half-open interval `[start, stop)` (in other words, the interval including *start* but excluding *stop*). For integer arguments the function is equivalent to the Python built-in range function, but returns a tensor rather than a list.

When using a non-integer step, such as 0.1, the results will often not be consistent. It is better to use `linspace` for these cases.

start [number, optional] Start of interval. The interval includes this value. The default start value is 0.

stop [number] End of interval. The interval does not include this value, except in some cases where *step* is not an integer and floating point round-off affects the length of *out*.

step [number, optional] Spacing between values. For any output *out*, this is the distance between two adjacent values, `out[i+1] - out[i]`. The default step size is 1. If *step* is specified as a position argument, *start* must also be given.

dtype [dtype] The type of the output tensor. If *dtype* is not given, infer the data type from the other input arguments.

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

arange [Tensor] Tensor of evenly spaced values.

For floating point arguments, the length of the result is `ceil((stop - start)/step)`. Because of floating point overflow, this rule may result in the last element of *out* being greater than *stop*.

linspace : Evenly spaced numbers with careful handling of endpoints. ogrid: Tensors of evenly spaced numbers in N-dimensions. mgrid: Grid-shaped tensors of evenly spaced numbers in N-dimensions.

```
>>> import mars.tensor as mt
```

```
>>> mt.arange(3).execute()
array([0, 1, 2])
>>> mt.arange(3.0).execute()
array([ 0.,  1.,  2.])
>>> mt.arange(3,7).execute()
array([3, 4, 5, 6])
>>> mt.arange(3,7,2).execute()
array([3, 5])
```

mars.tensor.linspace

mars.tensor.**linspace**(*start*, *stop*, *num=50*, *endpoint=True*, *retstep=False*, *dtype=None*, *gpu=False*, *chunk_size=None*)

Return evenly spaced numbers over a specified interval.

Returns *num* evenly spaced samples, calculated over the interval [*start*, *stop*].

The endpoint of the interval can optionally be excluded.

start [scalar] The starting value of the sequence.

stop [scalar] The end value of the sequence, unless *endpoint* is set to False. In that case, the sequence consists of all but the last of `num + 1` evenly spaced samples, so that *stop* is excluded. Note that the step size changes when *endpoint* is False.

num [int, optional] Number of samples to generate. Default is 50. Must be non-negative.

endpoint [bool, optional] If True, *stop* is the last sample. Otherwise, it is not included. Default is True.

retstep [bool, optional] If True, return (*samples*, *step*), where *step* is the spacing between samples.

dtype [dtype, optional] The type of the output tensor. If *dtype* is not given, infer the data type from the other input arguments.

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

samples [Tensor] There are *num* equally spaced samples in the closed interval `[start, stop]` or the half-open interval `[start, stop)` (depending on whether *endpoint* is True or False).

step [float, optional] Only returned if *retstep* is True

Size of spacing between samples.

arange [Similar to *linspace*, but uses a step size (instead of the] number of samples).

logspace : Samples uniformly distributed in log space.

```
>>> import mars.tensor as mt
>>> from mars.session import new_session
```

```
>>> sess = new_session().as_default()
```

```
>>> mt.linspace(2.0, 3.0, num=5).execute()
array([ 2.  ,  2.25,  2.5 ,  2.75,  3.  ])
>>> mt.linspace(2.0, 3.0, num=5, endpoint=False).execute()
array([ 2. ,  2.2,  2.4,  2.6,  2.8])
>>> sess.run(mt.linspace(2.0, 3.0, num=5, retstep=True).execute())
(array([ 2.  ,  2.25,  2.5 ,  2.75,  3.  ]), 0.25)
```

Graphical illustration:

```
>>> import matplotlib.pyplot as plt
>>> N = 8
>>> y = mt.zeros(N)
>>> x1 = mt.linspace(0, 10, N, endpoint=True)
>>> x2 = mt.linspace(0, 10, N, endpoint=False)
>>> plt.plot(x1.execute(), y.execute(), 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.plot(x2.execute(), y.execute() + 0.5, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.ylim([-0.5, 1])
(-0.5, 1)
>>> plt.show()
```

mars.tensor.meshgrid

mars.tensor.**meshgrid**(**xi*, ***kwargs*)

Return coordinate matrices from coordinate vectors.

Make N-D coordinate arrays for vectorized evaluations of N-D scalar/vector fields over N-D grids, given one-dimensional coordinate tensors x1, x2,..., xn.

x1, x2,..., xn [array_like] 1-D arrays representing the coordinates of a grid.

indexing [{'xy', 'ij'}, optional] Cartesian ('xy', default) or matrix ('ij') indexing of output. See Notes for more details.

sparse [bool, optional] If True a sparse grid is returned in order to conserve memory. Default is False.

X1, X2,..., XN [Tensor] For vectors *x1*, *x2*,..., 'xn' with lengths `Ni=len(xi)` , return `(N1, N2, N3,...Nn)` shaped tensors if indexing='ij' or `(N2, N1, N3,...Nn)` shaped tensors if indexing='xy' with the elements of *xi* repeated to fill the matrix along the first dimension for *x1*, the second for *x2* and so on.

This function supports both indexing conventions through the indexing keyword argument. Giving the string 'ij' returns a meshgrid with matrix indexing, while 'xy' returns a meshgrid with Cartesian indexing. In the 2-D case with inputs of length M and N, the outputs are of shape (N, M) for 'xy' indexing and (M, N) for 'ij' indexing. In the 3-D case with inputs of length M, N and P, outputs are of shape (N, M, P) for 'xy' indexing and (M, N, P) for 'ij' indexing. The difference is illustrated by the following code snippet:

```
xv, yv = mt.meshgrid(x, y, sparse=False, indexing='ij')
for i in range(nx):
    for j in range(ny):
        # treat xv[i,j], yv[i,j]

xv, yv = mt.meshgrid(x, y, sparse=False, indexing='xy')
for i in range(nx):
    for j in range(ny):
        # treat xv[j,i], yv[j,i]
```

In the 1-D and 0-D case, the indexing and sparse keywords have no effect.

```
>>> import mars.tensor as mt
```

```
>>> nx, ny = (3, 2)
>>> x = mt.linspace(0, 1, nx)
>>> y = mt.linspace(0, 1, ny)
>>> xv, yv = mt.meshgrid(x, y)
>>> xv.execute()
array([[ 0. ,  0.5,  1. ],
       [ 0. ,  0.5,  1. ]])
>>> yv.execute()
array([[ 0.,  0.,  0.],
       [ 1.,  1.,  1.]])
>>> xv, yv = mt.meshgrid(x, y, sparse=True)  # make sparse output arrays
>>> xv.execute()
array([[ 0. ,  0.5,  1. ]])
>>> yv.execute()
array([[ 0.],
       [ 1.]])
```

meshgrid is very useful to evaluate functions on a grid.

```
>>> import matplotlib.pyplot as plt
>>> x = mt.arange(-5, 5, 0.1)
>>> y = mt.arange(-5, 5, 0.1)
>>> xx, yy = mt.meshgrid(x, y, sparse=True)
>>> z = mt.sin(xx**2 + yy**2) / (xx**2 + yy**2)
>>> h = plt.contourf(x,y,z)
```

mars.tensor.mgrid

mars.tensor.**mgrid = <mars.tensor.lib.index_tricks.nd_grid object>**

Construct a multi-dimensional "meshgrid".

`grid = nd_grid()` creates an instance which will return a mesh-grid when indexed. The dimension and number of the output arrays are equal to the number of indexing dimensions. If the step length is not a complex number, then the stop is not inclusive.

However, if the step length is a **complex number** (e.g. 5j), then the integer part of its magnitude is interpreted as specifying the number of points to create between the start and stop values, where the stop value **is inclusive**.

If instantiated with an argument of `sparse=True`, the mesh-grid is open (or not fleshed out) so that only one-dimension of each returned argument is greater than 1.

sparse [bool, optional] Whether the grid is sparse or not. Default is False.

Two instances of *nd_grid* are made available in the Mars.tensor namespace, *mgrid* and *ogrid*:

```
mgrid = nd_grid(sparse=False)
ogrid = nd_grid(sparse=True)
```

Users should use these pre-defined instances instead of using *nd_grid* directly.

```
>>> import mars.tensor as mt
```

```
>>> mgrid = mt.lib.index_tricks.nd_grid()
>>> mgrid[0:5,0:5]
array([[[0, 0, 0, 0, 0],
        [1, 1, 1, 1, 1],
        [2, 2, 2, 2, 2],
        [3, 3, 3, 3, 3],
        [4, 4, 4, 4, 4]],
       [[0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4]]])
>>> mgrid[-1:1:5j]
array([-1. , -0.5,  0. ,  0.5,  1. ])
```

```
>>> ogrid = mt.lib.index_tricks.nd_grid(sparse=True)
>>> ogrid[0:5,0:5]
[array([[0],
        [1],
        [2],
        [3],
        [4]]), array([[0, 1, 2, 3, 4]])]
```

mars.tensor.ogrid

mars.tensor.**ogrid = <mars.tensor.lib.index_tricks.nd_grid object>**

Construct a multi-dimensional "meshgrid".

`grid = nd_grid()` creates an instance which will return a mesh-grid when indexed. The dimension and number of the output arrays are equal to the number of indexing dimensions. If the step length is not a complex number, then the stop is not inclusive.

However, if the step length is a **complex number** (e.g. 5j), then the integer part of its magnitude is interpreted as specifying the number of points to create between the start and stop values, where the stop value **is inclusive**.

If instantiated with an argument of `sparse=True`, the mesh-grid is open (or not fleshed out) so that only one-dimension of each returned argument is greater than 1.

sparse [bool, optional] Whether the grid is sparse or not. Default is False.

Two instances of *nd_grid* are made available in the Mars.tensor namespace, *mgrid* and *ogrid*:

```
mgrid = nd_grid(sparse=False)
ogrid = nd_grid(sparse=True)
```

Users should use these pre-defined instances instead of using *nd_grid* directly.

```
>>> import mars.tensor as mt
```

```
>>> mgrid = mt.lib.index_tricks.nd_grid()
>>> mgrid[0:5,0:5]
array([[[0, 0, 0, 0, 0],
        [1, 1, 1, 1, 1],
        [2, 2, 2, 2, 2],
        [3, 3, 3, 3, 3],
        [4, 4, 4, 4, 4]],
       [[0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4]]])
>>> mgrid[-1:1:5j]
array([-1. , -0.5,  0. ,  0.5,  1. ])
```

```
>>> ogrid = mt.lib.index_tricks.nd_grid(sparse=True)
>>> ogrid[0:5,0:5]
[array([[0],
        [1],
        [2],
        [3],
        [4]]), array([[0, 1, 2, 3, 4]])]
```

Building matrices

mars.tensor.diag	Extract a diagonal or construct a diagonal tensor.
mars.tensor.diagflat	Create a two-dimensional tensor with the flattened input as a diagonal.
mars.tensor.tril	Lower triangle of a tensor.

Continued on next page

Table 10 – continued from previous page

mars.tensor.triu	Upper triangle of a tensor.

mars.tensor.diag

mars.tensor.**diag**(*v*, *k=0*, *sparse=None*, *gpu=False*, *chunk_size=None*)

Extract a diagonal or construct a diagonal tensor.

See the more detailed documentation for `mt.diagonal` if you use this function to extract a diagonal and wish to write to the resulting tensor

v [array_like] If *v* is a 2-D tensor, return its *k*-th diagonal. If *v* is a 1-D tensor, return a 2-D tensor with *v* on the *k*-th diagonal.

k [int, optional] Diagonal in question. The default is 0. Use *k>0* for diagonals above the main diagonal, and *k<0* for diagonals below the main diagonal.

sparse: bool, optional Create sparse tensor if True, False as default

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

out [Tensor] The extracted diagonal or constructed diagonal tensor.

diagonal : Return specified diagonals. diagflat : Create a 2-D array with the flattened input as a diagonal. trace : Sum along diagonals. triu : Upper triangle of a tensor. tril : Lower triangle of a tensor.

```
>>> import mars.tensor as mt
```

```
>>> x = mt.arange(9).reshape((3,3))
>>> x.execute()
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

```
>>> mt.diag(x).execute()
array([0, 4, 8])
>>> mt.diag(x, k=1).execute()
array([1, 5])
>>> mt.diag(x, k=-1).execute()
array([3, 7])
```

```
>>> mt.diag(mt.diag(x)).execute()
array([[0, 0, 0],
       [0, 4, 0],
       [0, 0, 8]])
```

mars.tensor.diagflat

mars.tensor.**diagflat**(*v*, *k=0*, *sparse=None*, *gpu=False*, *chunk_size=None*)

Create a two-dimensional tensor with the flattened input as a diagonal.

v [array_like] Input data, which is flattened and set as the *k*-th diagonal of the output.

k [int, optional] Diagonal to set; 0, the default, corresponds to the "main" diagonal, a positive (negative) *k* giving the number of the diagonal above (below) the main.

sparse: bool, optional Create sparse tensor if True, False as default

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

out [Tensor] The 2-D output tensor.

diag : MATLAB work-alike for 1-D and 2-D tensors. diagonal : Return specified diagonals. trace : Sum along diagonals.

```
>>> import mars.tensor as mt
```

```
>>> mt.diagflat([[1,2], [3,4]]).execute()
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

```
>>> mt.diagflat([1,2], 1).execute()
array([[0, 1, 0],
       [0, 0, 2],
       [0, 0, 0]])
```

mars.tensor.tril

mars.tensor.**tril**(*m*, *k=0*, *gpu=None*)

Lower triangle of a tensor.

Return a copy of a tensor with elements above the *k*-th diagonal zeroed.

m [array_like, shape (M, N)] Input tensor.

k [int, optional] Diagonal above which to zero elements. *k = 0* (the default) is the main diagonal, *k < 0* is below it and *k > 0* is above.

gpu [bool, optional] Allocate the tensor on GPU if True, None as default

tril [Tensor, shape (M, N)] Lower triangle of *m*, of same shape and data-type as *m*.

triu : same thing, only for the upper triangle

```
>>> import mars.tensor as mt
```

```
>>> mt.tril([[1,2,3],[4,5,6],[7,8,9],[10,11,12]], -1).execute()
array([[ 0,  0,  0],
       [ 4,  0,  0],
       [ 7,  8,  0],
       [10, 11, 12]])
```

mars.tensor.triu

mars.tensor.**triu**(*m*, *k=0*, *gpu=None*)

Upper triangle of a tensor.

Return a copy of a matrix with the elements below the *k*-th diagonal zeroed.

Please refer to the documentation for *tril* for further details.

tril : lower triangle of a tensor

```
>>> import mars.tensor as mt
```

```
>>> mt.triu([[1,2,3],[4,5,6],[7,8,9],[10,11,12]], -1).execute()
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 0,  8,  9],
       [ 0,  0, 12]])
```

2.6.2 Tensor Manipulation Routines

Basic manipulations

mars.tensor.copyto	Copies values from one array to another, broadcasting as necessary.

mars.tensor.copyto

mars.tensor.**copyto**(*dst*, *src*, *casting='same_kind'*, *where=True*)

Copies values from one array to another, broadcasting as necessary.

Raises a TypeError if the *casting* rule is violated, and if *where* is provided, it selects which elements to copy.

dst [Tensor] The tensor into which values are copied.

src [array_like] The tensor from which values are copied.

casting [{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional] Controls what kind of data casting may occur when copying.

- 'no' means the data types should not be cast at all.
- 'equiv' means only byte-order changes are allowed.
- 'safe' means only casts which can preserve values are allowed.
- 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
- 'unsafe' means any data conversions may be done.

where [array_like of bool, optional] A boolean tensor which is broadcasted to match the dimensions of *dst*, and selects elements to copy from *src* to *dst* wherever it contains the value True.

Shape manipulation

mars.tensor.reshape	Gives a new shape to a tensor without changing its data.
mars.tensor.ravel	Return a contiguous flattened tensor.

mars.tensor.reshape

mars.tensor.**reshape**(*a*, *newshape*, *order='C'*)

Gives a new shape to a tensor without changing its data.

a [array_like] Tensor to be reshaped.

newshape [int or tuple of ints] The new shape should be compatible with the original shape. If an integer, then the result will be a 1-D tensor of that length. One shape dimension can be -1. In this case, the value is inferred from the length of the tensor and remaining dimensions.

order [{'C', 'F', 'A'}, optional] Read the elements of *a* using this index order, and place the elements into the reshaped array using this index order. 'C' means to read / write the elements using C-like index order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to read / write the elements using Fortran-like index order, with the first index changing fastest, and the last index changing slowest. Note that the 'C' and 'F' options take no account of the memory layout of the underlying array, and only refer to the order of indexing. 'A' means to read / write the elements in Fortran-like index order if *a* is Fortran *contiguous* in memory, C-like order otherwise.

reshaped_array [Tensor] This will be a new view object if possible; otherwise, it will be a copy.

Tensor.reshape : Equivalent method.

It is not always possible to change the shape of a tensor without copying the data. If you want an error to be raised when the data is copied, you should assign the new shape to the shape attribute of the array:

```
>>> import mars.tensor as mt
```

```
>>> a = mt.arange(6).reshape((3, 2))
>>> a.execute()
array([[0, 1],
       [2, 3],
       [4, 5]])
```

You can think of reshaping as first raveling the tensor (using the given index order), then inserting the elements from the raveled tensor into the new tensor using the same kind of index ordering as was used for the raveling.

```
>>> mt.reshape(a, (2, 3)).execute()
array([[0, 1, 2],
       [3, 4, 5]])
>>> mt.reshape(mt.ravel(a), (2, 3)).execute()
array([[0, 1, 2],
       [3, 4, 5]])
```

```
>>> a = mt.array([[1,2,3], [4,5,6]])
>>> mt.reshape(a, 6).execute()
array([1, 2, 3, 4, 5, 6])
```

```
>>> mt.reshape(a, (3,-1)).execute()       # the unspecified value is inferred to␣
↪be 2
array([[1, 2],
       [3, 4],
       [5, 6]])
```

mars.tensor.ravel

mars.tensor.**ravel**(*a*, *order='C'*)

Return a contiguous flattened tensor.

A 1-D tensor, containing the elements of the input, is returned. A copy is made only if needed.

a [array_like] Input tensor. The elements in *a* are packed as a 1-D tensor.

order : {'C','F', 'A', 'K'}, optional

The elements of *a* are read using this index order. 'C' means to index the elements in row-major, C-style order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to index the elements in column-major, Fortran-style order, with the first index changing fastest, and the last index changing slowest. Note that the 'C' and 'F' options take no account of the memory layout of the underlying array, and only refer to the order of axis indexing. 'A' means to read the elements in Fortran-like index order if *a* is Fortran *contiguous* in memory, C-like order otherwise. 'K' means to read the elements in the order they occur in memory, except for reversing the data when strides are negative. By default, 'C' index order is used.

y [array_like] If *a* is a matrix, y is a 1-D tensor, otherwise y is a tensor of the same subtype as *a*. The shape of the returned array is `(a.size,)`. Matrices are special cased for backward compatibility.

Tensor.flat : 1-D iterator over an array. Tensor.flatten : 1-D array copy of the elements of an array

in row-major order.

Tensor.reshape : Change the shape of an array without changing its data.

It is equivalent to `reshape(-1)`.

```
>>> import mars.tensor as mt
```

```
>>> x = mt.array([[1, 2, 3], [4, 5, 6]])
>>> print(mt.ravel(x).execute())
[1 2 3 4 5 6]
```

```
>>> print(x.reshape(-1).execute())
[1 2 3 4 5 6]
```

```
>>> print(mt.ravel(x.T).execute())
[1 4 2 5 3 6]
```

```
>>> a = mt.arange(12).reshape(2,3,2).swapaxes(1,2); a.execute()
array([[[ 0,  2,  4],
        [ 1,  3,  5]],
       [[ 6,  8, 10],
        [ 7,  9, 11]]])
>>> a.ravel().execute()
array([ 0,  2,  4,  1,  3,  5,  6,  8, 10,  7,  9, 11])
```

Transposition

mars.tensor.moveaxis	Move axes of a tensor to new positions.

Continued on next page

Table 13 – continued from previous page

mars.tensor.rollaxis	Roll the specified axis backwards, until it lies in a given position.
mars.tensor.swapaxes	Interchange two axes of a tensor.
mars.tensor.core.Tensor.T	Same as self.transpose(), except that self is returned if self.ndim < 2.
mars.tensor.transpose	Permute the dimensions of a tensor.

mars.tensor.moveaxis

mars.tensor.**moveaxis**(*a*, *source*, *destination*)

Move axes of a tensor to new positions.

Other axes remain in their original order.

a [Tensor] The tensor whose axes should be reordered.

source [int or sequence of int] Original positions of the axes to move. These must be unique.

destination [int or sequence of int] Destination positions for each of the original axes. These must also be unique.

result [Tensor] Array with moved axes. This tensor is a view of the input tensor.

transpose: Permute the dimensions of an array. swapaxes: Interchange two axes of an array.

```
>>> import mars.tensor as mt
```

```
>>> x = mt.zeros((3, 4, 5))
>>> mt.moveaxis(x, 0, -1).shape
(4, 5, 3)
>>> mt.moveaxis(x, -1, 0).shape
(5, 3, 4),
```

These all achieve the same result:

```
>>> mt.transpose(x).shape
(5, 4, 3)
>>> mt.swapaxes(x, 0, -1).shape
(5, 4, 3)
>>> mt.moveaxis(x, [0, 1], [-1, -2]).shape
(5, 4, 3)
>>> mt.moveaxis(x, [0, 1, 2], [-1, -2, -3]).shape
(5, 4, 3)
```

mars.tensor.rollaxis

mars.tensor.**rollaxis**(*tensor*, *axis*, *start=0*)

Roll the specified axis backwards, until it lies in a given position.

This function continues to be supported for backward compatibility, but you should prefer *moveaxis*.

a [Tensor] Input tensor.

axis [int] The axis to roll backwards. The positions of the other axes do not change relative to one another.

start [int, optional] The axis is rolled until it lies before this position. The default, 0, results in a "complete" roll.

res [Tensor] a view of *a* is always returned.

moveaxis : Move array axes to new positions. roll : Roll the elements of an array by a number of positions along a

given axis.

```
>>> import mars.tensor as mt
```

```
>>> a = mt.ones((3,4,5,6))
>>> mt.rollaxis(a, 3, 1).shape
(3, 6, 4, 5)
>>> mt.rollaxis(a, 2).shape
(5, 3, 4, 6)
>>> mt.rollaxis(a, 1, 4).shape
(3, 5, 6, 4)
```

mars.tensor.swapaxes

mars.tensor.**swapaxes**(*a*, *axis1*, *axis2*)

Interchange two axes of a tensor.

a [array_like] Input tensor.

axis1 [int] First axis.

axis2 [int] Second axis.

a_swapped [Tensor] If *a* is a Tensor, then a view of *a* is returned; otherwise a new tensor is created.

```
>>> import mars.tensor as mt
```

```
>>> x = mt.array([[1,2,3]])
>>> mt.swapaxes(x,0,1).execute()
array([[1],
       [2],
       [3]])
```

```
>>> x = mt.array([[[0,1],[2,3]],[[4,5],[6,7]]])
>>> x.execute()
array([[[0, 1],
        [2, 3]],
       [[4, 5],
        [6, 7]]])
```

```
>>> mt.swapaxes(x,0,2).execute()
array([[[0, 4],
        [2, 6]],
       [[1, 5],
        [3, 7]]])
```

mars.tensor.core.Tensor.T

Tensor.**T**

Same as self.transpose(), except that self is returned if self.ndim < 2.

```
>>> import mars.tensor as mt
```

```
>>> x = mt.array([[1.,2.],[3.,4.]])
>>> x.execute()
array([[ 1.,  2.],
       [ 3.,  4.]])
>>> x.T.execute()
array([[ 1.,  3.],
       [ 2.,  4.]])
>>> x = mt.array([1.,2.,3.,4.])
>>> x.execute()
array([ 1.,  2.,  3.,  4.])
>>> x.T.execute()
array([ 1.,  2.,  3.,  4.])
```

mars.tensor.transpose

mars.tensor.**transpose**(*a*, *axes=None*)

Permute the dimensions of a tensor.

a [array_like] Input tensor.

axes [list of ints, optional] By default, reverse the dimensions, otherwise permute the axes according to the values given.

p [Tensor] *a* with its axes permuted. A view is returned whenever possible.

moveaxis argsort

Use *transpose(a, argsort(axes))* to invert the transposition of tensors when using the *axes* keyword argument.

Transposing a 1-D array returns an unchanged view of the original tensor.

```
>>> import mars.tensor as mt
```

```
>>> x = mt.arange(4).reshape((2,2))
>>> x.execute()
array([[0, 1],
       [2, 3]])
```

```
>>> mt.transpose(x).execute()
array([[0, 2],
       [1, 3]])
```

```
>>> x = mt.ones((1, 2, 3))
>>> mt.transpose(x, (1, 0, 2)).shape
(2, 1, 3)
```

Edit dimensionalities

mars.tensor.atleast_1d	Convert inputs to tensors with at least one dimension.
mars.tensor.atleast_2d	View inputs as tensors with at least two dimensions.
mars.tensor.atleast_3d	View inputs as tensors with at least three dimensions.
mars.tensor.broadcast_to	Broadcast an tensor to a new shape.
mars.tensor.broadcast_arrays	Broadcast any number of arrays against each other.
mars.tensor.expand_dims	Expand the shape of a tensor.
mars.tensor.squeeze	Remove single-dimensional entries from the shape of a tensor.

mars.tensor.atleast_1d

mars.tensor.**atleast_1d**(**tensors*)

Convert inputs to tensors with at least one dimension.

Scalar inputs are converted to 1-dimensional tensors, whilst higher-dimensional inputs are preserved.

tensors1, tensors2, ... [array_like] One or more input tensors.

ret [Tensor] An tensor, or list of tensors, each with `a.ndim >= 1`. Copies are made only if necessary.

atleast_2d, atleast_3d

```
>>> import mars.tensor as mt
>>> from mars.session import new_session
```

```
>>> sess = new_session().as_default()
```

```
>>> mt.atleast_1d(1.0).execute()
array([ 1.])
```

```
>>> x = mt.arange(9.0).reshape(3,3)
>>> mt.atleast_1d(x).execute()
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 6.,  7.,  8.]])
>>> mt.atleast_1d(x) is x
True
```

```
>>> sess.run(mt.atleast_1d(1, [3, 4]))
[array([1]), array([3, 4])]
```

mars.tensor.atleast_2d

mars.tensor.**atleast_2d**(**tensors*)

View inputs as tensors with at least two dimensions.

tensors1, tensors2, ... [array_like] One or more array-like sequences. Non-tensor inputs are converted to tensors. Tensors that already have two or more dimensions are preserved.

res, res2, ... [Tensor] A tensor, or list of tensors, each with `a.ndim >= 2`. Copies are avoided where possible, and views with two or more dimensions are returned.

atleast_1d, atleast_3d

```
>>> import mars.tensor as mt
>>> from mars.session import new_session
```

```
>>> sess = new_session().as_default()
```

```
>>> mt.atleast_2d(3.0).execute()
array([[ 3.]])
```

```
>>> x = mt.arange(3.0)
>>> mt.atleast_2d(x).execute()
array([[ 0.,  1.,  2.]])
```

```
>>> sess.run(mt.atleast_2d(1, [1, 2], [[1, 2]]))
[array([[1]]), array([[1, 2]]), array([[1, 2]])]
```

mars.tensor.atleast_3d

mars.tensor.**atleast_3d**(**tensors*)

View inputs as tensors with at least three dimensions.

tensors1, tensors2, ... [array_like] One or more tensor-like sequences. Non-tensor inputs are converted to tensors. Tensors that already have three or more dimensions are preserved.

res1, res2, ... [Tensor] A tensor, or list of tensors, each with `a.ndim >= 3`. Copies are avoided where possible, and views with three or more dimensions are returned. For example, a 1-D tensor of shape `(N,)` becomes a view of shape `(1, N, 1)`, and a 2-D tensor of shape `(M, N)` becomes a view of shape `(M, N, 1)`.

atleast_1d, atleast_2d

```
>>> import mars.tensor as mt
>>> from mars.session import new_session
```

```
>>> sess = new_session()
```

```
>>> mt.atleast_3d(3.0).execute()
array([[[ 3.]]])
```

```
>>> x = mt.arange(3.0)
>>> mt.atleast_3d(x).shape
(1, 3, 1)
```

```
>>> x = mt.arange(12.0).reshape(4,3)
>>> mt.atleast_3d(x).shape
(4, 3, 1)
```

```
>>> for arr in sess.run(mt.atleast_3d([1, 2], [[1, 2]], [[[1, 2]]])):
...     print(arr, arr.shape)
...
[[[1]
```

(continues on next page)

(continued from previous page)

```
  [2]]] (1, 2, 1)
[[[1]
  [2]]] (1, 2, 1)
[[[1 2]]] (1, 1, 2)
```

mars.tensor.broadcast_to

mars.tensor.**broadcast_to**(*tensor*, *shape*)

Broadcast an tensor to a new shape.

tensor [array_like] The tensor to broadcast.

shape [tuple] The shape of the desired array.

broadcast : Tensor

ValueError If the tensor is not compatible with the new shape according to Mars's broadcasting rules.

```
>>> import mars.tensor as mt
```

```
>>> x = mt.array([1, 2, 3])
>>> mt.broadcast_to(x, (3, 3)).execute()
array([[1, 2, 3],
       [1, 2, 3],
       [1, 2, 3]])
```

mars.tensor.broadcast_arrays

mars.tensor.**broadcast_arrays**(**args*, ***kwargs*)

Broadcast any number of arrays against each other.

****args*** [array_likes] The tensors to broadcast.

broadcasted : list of tensors

```
>>> import mars.tensor as mt
>>> from mars.session import new_session
```

```
>>> sess = new_session().as_default()
>>> x = mt.array([[1,2,3]])
>>> y = mt.array([[1],[2],[3]])
>>> sess.run(mt.broadcast_arrays(x, y))
[array([[1, 2, 3],
       [1, 2, 3],
       [1, 2, 3]]), array([[1, 1, 1],
       [2, 2, 2],
       [3, 3, 3]])]
```

mars.tensor.expand_dims

mars.tensor.**expand_dims**(*a*, *axis*)

Expand the shape of a tensor.

Insert a new axis that will appear at the *axis* position in the expanded array shape.

a [array_like] Input tensor.

axis [int] Position in the expanded axes where the new axis is placed.

res [Tensor] Output tensor. The number of dimensions is one greater than that of the input tensor.

squeeze : The inverse operation, removing singleton dimensions reshape : Insert, remove, and combine dimensions, and resize existing ones doc.indexing, atleast_1d, atleast_2d, atleast_3d

```
>>> import mars.tensor as mt
```

```
>>> x = mt.array([1,2])
>>> x.shape
(2,)
```

The following is equivalent to `x[mt.newaxis,:]` or `x[mt.newaxis]`:

```
>>> y = mt.expand_dims(x, axis=0)
>>> y.execute()
array([[1, 2]])
>>> y.shape
(1, 2)
```

```
>>> y = mt.expand_dims(x, axis=1)  # Equivalent to x[:,mt.newaxis]
>>> y.execute()
array([[1],
       [2]])
>>> y.shape
(2, 1)
```

Note that some examples may use `None` instead of `np.newaxis`. These are the same objects:

```
>>> mt.newaxis is None
True
```

mars.tensor.squeeze

mars.tensor.**squeeze**(*a*, *axis=None*)

Remove single-dimensional entries from the shape of a tensor.

a [array_like] Input data.

axis [None or int or tuple of ints, optional] Selects a subset of the single-dimensional entries in the shape. If an axis is selected with shape entry greater than one, an error is raised.

squeezed [Tensor] The input tensor, but with all or a subset of the dimensions of length 1 removed. This is always *a* itself or a view into *a*.

ValueError If *axis* is not *None*, and an axis being squeezed is not of length 1

expand_dims : The inverse operation, adding singleton dimensions reshape : Insert, remove, and combine dimensions, and resize existing ones

```
>>> import mars.tensor as mt
```

```
>>> x = mt.array([[[0], [1], [2]]])
>>> x.shape
(1, 3, 1)
>>> mt.squeeze(x).shape
(3,)
>>> mt.squeeze(x, axis=0).shape
(3, 1)
>>> mt.squeeze(x, axis=1).shape
Traceback (most recent call last):
...
ValueError: cannot select an axis to squeeze out which has size not equal to one
>>> mt.squeeze(x, axis=2).shape
(1, 3)
```

Changing kind of tensor

mars.tensor.asarray	Convert the input to an array.

Joining tensors

mars.tensor.concatenate	Join a sequence of arrays along an existing axis.
mars.tensor.stack	Join a sequence of tensors along a new axis.
mars.tensor.column_stack	Stack 1-D tensors as columns into a 2-D tensor.
mars.tensor.dstack	Stack tensors in sequence depth wise (along third axis).
mars.tensor.hstack	Stack tensors in sequence horizontally (column wise).
mars.tensor.vstack	Stack tensors in sequence vertically (row wise).

mars.tensor.concatenate

mars.tensor.**concatenate**(*tensors*, *axis=0*)

Join a sequence of arrays along an existing axis.

a1, a2, . . . [sequence of array_like] The tensors must have the same shape, except in the dimension corresponding to *axis* (the first, by default).

axis [int, optional] The axis along which the tensors will be joined. Default is 0.

res [Tensor] The concatenated tensor.

array_split [Split a tensor into multiple sub-arrays of equal or] near-equal size.

split : Split tensor into a list of multiple sub-tensors of equal size. hsplit : Split tensor into multiple sub-tensors horizontally (column wise) vsplit : Split tensor into multiple sub-tensors vertically (row wise) dsplit : Split tensor into multiple sub-tensors along the 3rd axis (depth). stack : Stack a sequence of tensors along a new axis. hstack : Stack tensors in sequence horizontally (column wise) vstack : Stack tensors in sequence vertically (row wise) dstack : Stack tensors in sequence depth wise (along third dimension)

```
>>> import mars.tensor as mt
```

```
>>> a = mt.array([[1, 2], [3, 4]])
>>> b = mt.array([[5, 6]])
```

(continues on next page)

(continued from previous page)

```
>>> mt.concatenate((a, b), axis=0).execute()
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> mt.concatenate((a, b.T), axis=1).execute()
array([[1, 2, 5],
       [3, 4, 6]])
```

mars.tensor.stack

mars.tensor.**stack**(*tensors*, *axis=0*, *out=None*)

Join a sequence of tensors along a new axis.

The *axis* parameter specifies the index of the new axis in the dimensions of the result. For example, if `axis=0` it will be the first dimension and if `axis=-1` it will be the last dimension.

tensors [sequence of array_like] Each tensor must have the same shape.

axis [int, optional] The axis in the result tensor along which the input tensors are stacked.

out [Tensor, optional] If provided, the destination to place the result. The shape must be correct, matching that of what stack would have returned if no out argument were specified.

stacked [Tensor] The stacked tensor has one more dimension than the input tensors.

concatenate : Join a sequence of tensors along an existing axis. split : Split tensor into a list of multiple sub-tensors of equal size. block : Assemble tensors from blocks.

```
>>> import mars.tensor as mt
```

```
>>> arrays = [mt.random.randn(3, 4) for _ in range(10)]
>>> mt.stack(arrays, axis=0).shape
(10, 3, 4)
```

```
>>> mt.stack(arrays, axis=1).shape
(3, 10, 4)
```

```
>>> mt.stack(arrays, axis=2).shape
(3, 4, 10)
```

```
>>> a = mt.array([1, 2, 3])
>>> b = mt.array([2, 3, 4])
>>> mt.stack((a, b)).execute()
array([[1, 2, 3],
       [2, 3, 4]])
```

```
>>> mt.stack((a, b), axis=-1).execute()
array([[1, 2],
       [2, 3],
       [3, 4]])
```

mars.tensor.column_stack

mars.tensor.**column_stack**(*tup*)

Stack 1-D tensors as columns into a 2-D tensor.

Take a sequence of 1-D tensors and stack them as columns to make a single 2-D tensor. 2-D tensors are stacked as-is, just like with *hstack*. 1-D tensors are turned into 2-D columns first.

tup [sequence of 1-D or 2-D tensors.] Tensors to stack. All of them must have the same first dimension.

stacked [2-D tensor] The tensor formed by stacking the given tensors.

stack, hstack, vstack, concatenate

```
>>> import mars.tensor as mt
```

```
>>> a = mt.array((1,2,3))
>>> b = mt.array((2,3,4))
>>> mt.column_stack((a,b)).execute()
array([[1, 2],
       [2, 3],
       [3, 4]])
```

mars.tensor.dstack

mars.tensor.**dstack**(*tup*)

Stack tensors in sequence depth wise (along third axis).

This is equivalent to concatenation along the third axis after 2-D tensors of shape *(M,N)* have been reshaped to *(M,N,1)* and 1-D arrays of shape *(N,)* have been reshaped to *(1,N,1)*. Rebuilds arrays divided by *dsplit*.

This function makes most sense for arrays with up to 3 dimensions. For instance, for pixel-data with a height (first axis), width (second axis), and r/g/b channels (third axis). The functions *concatenate*, *stack* and *block* provide more general stacking and concatenation operations.

tup [sequence of tensors] The tensors must have the same shape along all but the third axis. 1-D or 2-D arrays must have the same shape.

stacked [Tensor] The array formed by stacking the given tensors, will be at least 3-D.

stack : Join a sequence of tensors along a new axis. vstack : Stack along first axis. hstack : Stack along second axis. concatenate : Join a sequence of arrays along an existing axis. dsplit : Split tensor along third axis.

```
>>> import mars.tensor as mt
```

```
>>> a = mt.array((1,2,3))
>>> b = mt.array((2,3,4))
>>> mt.dstack((a,b)).execute()
array([[[1, 2],
        [2, 3],
        [3, 4]]])
```

```
>>> a = mt.array([[1],[2],[3]])
>>> b = mt.array([[2],[3],[4]])
>>> mt.dstack((a,b)).execute()
```

(continues on next page)

(continued from previous page)

```
array([[[1, 2]],
       [[2, 3]],
       [[3, 4]]])
```

mars.tensor.hstack

mars.tensor.**hstack**(*tup*)

Stack tensors in sequence horizontally (column wise).

This is equivalent to concatenation along the second axis, except for 1-D tensors where it concatenates along the first axis. Rebuilds tensors divided by *hsplit*.

This function makes most sense for tensors with up to 3 dimensions. For instance, for pixel-data with a height (first axis), width (second axis), and r/g/b channels (third axis). The functions *concatenate*, *stack* and *block* provide more general stacking and concatenation operations.

tup [sequence of tensors] The tensors must have the same shape along all but the second axis, except 1-D tensors which can be any length.

stacked [Tensor] The tensor formed by stacking the given tensors.

stack : Join a sequence of tensors along a new axis. vstack : Stack tensors in sequence vertically (row wise). dstack : Stack tensors in sequence depth wise (along third axis). concatenate : Join a sequence of tensors along an existing axis. hsplit : Split tensor along second axis. block : Assemble tensors from blocks.

```
>>> import mars.tensor as mt
```

```
>>> a = mt.array((1,2,3))
>>> b = mt.array((2,3,4))
>>> mt.hstack((a,b)).execute()
array([1, 2, 3, 2, 3, 4])
>>> a = mt.array([[1],[2],[3]])
>>> b = mt.array([[2],[3],[4]])
>>> mt.hstack((a,b)).execute()
array([[1, 2],
       [2, 3],
       [3, 4]])
```

mars.tensor.vstack

mars.tensor.**vstack**(*tup*)

Stack tensors in sequence vertically (row wise).

This is equivalent to concatenation along the first axis after 1-D tensors of shape *(N,)* have been reshaped to *(1,N)*. Rebuilds tensors divided by *vsplit*.

This function makes most sense for tensors with up to 3 dimensions. For instance, for pixel-data with a height (first axis), width (second axis), and r/g/b channels (third axis). The functions *concatenate*, *stack* and *block* provide more general stacking and concatenation operations.

tup [sequence of tensors] The tensors must have the same shape along all but the first axis. 1-D tensors must have the same length.

stacked [Tensor] The tensor formed by stacking the given tensors, will be at least 2-D.

stack : Join a sequence of tensors along a new axis. hstack : Stack tensors in sequence horizontally (column wise). dstack : Stack tensors in sequence depth wise (along third dimension). concatenate : Join a sequence of tensors along an existing axis. vsplit : Split tensor into a list of multiple sub-arrays vertically. block : Assemble tensors from blocks.

```
>>> import mars.tensor as mt
```

```
>>> a = mt.array([1, 2, 3])
>>> b = mt.array([2, 3, 4])
>>> mt.vstack((a,b)).execute()
array([[1, 2, 3],
       [2, 3, 4]])
```

```
>>> a = mt.array([[1], [2], [3]])
>>> b = mt.array([[2], [3], [4]])
>>> mt.vstack((a,b)).execute()
array([[1],
       [2],
       [3],
       [2],
       [3],
       [4]])
```

Splitting tensors

mars.tensor.split	Split a tensor into multiple sub-tensors.
mars.tensor.array_split	Split a tensor into multiple sub-tensors.
mars.tensor.dsplit	Split tensor into multiple sub-tensors along the 3rd axis (depth).
mars.tensor.hsplit	Split a tensor into multiple sub-tensors horizontally (column-wise).
mars.tensor.vsplit	Split a tensor into multiple sub-tensors vertically (row-wise).

mars.tensor.split

mars.tensor.**split**(*ary*, *indices_or_sections*, *axis=0*)

Split a tensor into multiple sub-tensors.

ary [Tensor] Tensor to be divided into sub-tensors.

indices_or_sections [int or 1-D tensor] If *indices_or_sections* is an integer, N, the array will be divided into N equal tensors along *axis*. If such a split is not possible, an error is raised.

If *indices_or_sections* is a 1-D tensor of sorted integers, the entries indicate where along *axis* the array is split. For example, `[2, 3]` would, for `axis=0`, result in

- ary[:2]
- ary[2:3]
- ary[3:]

If an index exceeds the dimension of the tensor along *axis*, an empty sub-tensor is returned correspondingly.

axis [int, optional] The axis along which to split, default is 0.

sub-tensors [list of Tensors] A list of sub-tensors.

ValueError If *indices_or_sections* is given as an integer, but a split does not result in equal division.

array_split [Split a tensor into multiple sub-tensors of equal or] near-equal size. Does not raise an exception if an equal division cannot be made.

hsplit : Split into multiple sub-arrays horizontally (column-wise). vsplit : Split tensor into multiple sub-tensors vertically (row wise). dsplit : Split tensor into multiple sub-tensors along the 3rd axis (depth). concatenate : Join a sequence of tensors along an existing axis. stack : Join a sequence of tensors along a new axis. hstack : Stack tensors in sequence horizontally (column wise). vstack : Stack tensors in sequence vertically (row wise). dstack : Stack tensors in sequence depth wise (along third dimension).

```
>>> import mars.tensor as mt
>>> from mars.session import new_session
```

```
>>> sess = new_session().as_default()
```

```
>>> x = mt.arange(9.0)
>>> sess.run(mt.split(x, 3))
[array([ 0.,  1.,  2.]), array([ 3.,  4.,  5.]), array([ 6.,  7.,  8.])]
```

```
>>> x = mt.arange(8.0)
>>> sess.run(mt.split(x, [3, 5, 6, 10]))
[array([ 0.,  1.,  2.]),
 array([ 3.,  4.]),
 array([ 5.]),
 array([ 6.,  7.]),
 array([], dtype=float64)]
```

mars.tensor.array_split

mars.tensor.**array_split**(*a*, *indices_or_sections*, *axis=0*)

Split a tensor into multiple sub-tensors.

Please refer to the `split` documentation. The only difference between these functions is that `array_split` allows *indices_or_sections* to be an integer that does *not* equally divide the axis. For a tensor of length l that should be split into n sections, it returns l % n sub-arrays of size l//n + 1 and the rest of size l//n.

split : Split tensor into multiple sub-tensors of equal size.

```
>>> import mars.tensor as mt
>>> from mars.session import new_session
```

```
>>> sess = new_session().as_default()
```

```
>>> x = mt.arange(8.0)
>>> sess.run(mt.array_split(x, 3))
    [array([ 0.,  1.,  2.]), array([ 3.,  4.,  5.]), array([ 6.,  7.])]
```

```
>>> x = mt.arange(7.0)
>>> sess.run(mt.array_split(x, 3))
    [array([ 0.,  1.,  2.]), array([ 3.,  4.]), array([ 5.,  6.])]
```

mars.tensor.dsplit

mars.tensor.**dsplit**(*a*, *indices_or_sections*)

Split tensor into multiple sub-tensors along the 3rd axis (depth).

Please refer to the *split* documentation. *dsplit* is equivalent to *split* with `axis=2`, the array is always split along the third axis provided the tensor dimension is greater than or equal to 3.

split : Split a tensor into multiple sub-arrays of equal size.

```
>>> import mars.tensor as mt
>>> from mars.session import new_session
```

```
>>> sess = new_session().as_default()
```

```
>>> x = mt.arange(16.0).reshape(2, 2, 4)
>>> x.execute()
array([[[  0.,   1.,   2.,   3.],
        [  4.,   5.,   6.,   7.]],
       [[  8.,   9.,  10.,  11.],
        [ 12.,  13.,  14.,  15.]]])
>>> sess.run(mt.dsplit(x, 2))
[array([[[  0.,   1.],
        [  4.,   5.]],
       [[  8.,   9.],
        [ 12.,  13.]]]),
 array([[[  2.,   3.],
        [  6.,   7.]],
       [[ 10.,  11.],
        [ 14.,  15.]]])]
>>> sess.run(mt.dsplit(x, mt.array([3, 6])))
[array([[[  0.,   1.,   2.],
        [  4.,   5.,   6.]],
       [[  8.,   9.,  10.],
        [ 12.,  13.,  14.]]]),
 array([[[  3.],
        [  7.]],
       [[ 11.],
        [ 15.]]]),
 array([], dtype=float64)]
```

mars.tensor.hsplit

mars.tensor.**hsplit**(*a*, *indices_or_sections*)

Split a tensor into multiple sub-tensors horizontally (column-wise).

Please refer to the *split* documentation. *hsplit* is equivalent to *split* with `axis=1`, the tensor is always split along the second axis regardless of the tensor dimension.

split : Split an array into multiple sub-arrays of equal size.

```
>>> import mars.tensor as mt
>>> from mars.session import new_session
```

```
>>> sess = new_session().as_default()
```

```
>>> x = mt.arange(16.0).reshape(4, 4)
>>> x.execute()
array([[  0.,   1.,   2.,   3.],
       [  4.,   5.,   6.,   7.],
       [  8.,   9.,  10.,  11.],
       [ 12.,  13.,  14.,  15.]])
>>> sess.run(mt.hsplit(x, 2))
[array([[  0.,   1.],
       [  4.,   5.],
       [  8.,   9.],
       [ 12.,  13.]]),
 array([[  2.,   3.],
       [  6.,   7.],
       [ 10.,  11.],
       [ 14.,  15.]])]
>>> sess.run(mt.hsplit(x, mt.array([3, 6])))
[array([[  0.,   1.,   2.],
       [  4.,   5.,   6.],
       [  8.,   9.,  10.],
       [ 12.,  13.,  14.]]),
 array([[  3.],
       [  7.],
       [ 11.],
       [ 15.]]),
 array([], dtype=float64)]
```

With a higher dimensional array the split is still along the second axis.

```
>>> x = mt.arange(8.0).reshape(2, 2, 2)
>>> x.execute()
array([[[ 0.,  1.],
        [ 2.,  3.]],
       [[ 4.,  5.],
        [ 6.,  7.]]])
>>> mt.hsplit(x, 2)
[array([[[ 0.,  1.]],
       [[ 4.,  5.]]]),
 array([[[ 2.,  3.]],
       [[ 6.,  7.]]])]
```

mars.tensor.vsplit

mars.tensor.**vsplit**(*a*, *indices_or_sections*)

Split a tensor into multiple sub-tensors vertically (row-wise).

Please refer to the `split` documentation. `vsplit` is equivalent to `split` with *axis=0* (default), the tensor is always split along the first axis regardless of the tensor dimension.

split : Split a tensor into multiple sub-tensors of equal size.

```
>>> import mars.tensor as mt
>>> from mars.session import new_session
```

```
>>> sess = new_session()
```

```
>>> x = mt.arange(16.0).reshape(4, 4)
>>> x.execute()
array([[  0.,   1.,   2.,   3.],
       [  4.,   5.,   6.,   7.],
       [  8.,   9.,  10.,  11.],
       [ 12.,  13.,  14.,  15.]])
>>> sess.run(mt.vsplit(x, 2))
[array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.]]),
 array([[  8.,   9.,  10.,  11.],
       [ 12.,  13.,  14.,  15.]])]
>>> sess.run(mt.vsplit(x, mt.array([3, 6])))
[array([[  0.,   1.,   2.,   3.],
       [  4.,   5.,   6.,   7.],
       [  8.,   9.,  10.,  11.]]),
 array([[ 12.,  13.,  14.,  15.]]),
 array([], dtype=float64)]
```

With a higher dimensional tensor the split is still along the first axis.

```
>>> x = mt.arange(8.0).reshape(2, 2, 2)
>>> x.execute()
array([[[ 0.,  1.],
        [ 2.,  3.]],
       [[ 4.,  5.],
        [ 6.,  7.]]])
>>> sess.run(mt.vsplit(x, 2))
[array([[[ 0.,  1.],
        [ 2.,  3.]]]),
 array([[[ 4.,  5.],
        [ 6.,  7.]]])]
```

Tiling tensors

mars.tensor.tile	Construct a tensor by repeating A the number of times given by reps.
mars.tensor.repeat	Repeat elements of a tensor.

mars.tensor.tile

mars.tensor.**tile**(*A*, *reps*)

Construct a tensor by repeating A the number of times given by reps.

If *reps* has length `d`, the result will have dimension of `max(d, A.ndim)`.

If `A.ndim < d`, *A* is promoted to be d-dimensional by prepending new axes. So a shape (3,) array is promoted to (1, 3) for 2-D replication, or shape (1, 1, 3) for 3-D replication. If this is not the desired behavior, promote *A* to d-dimensions manually before calling this function.

If `A.ndim < d`, *reps* is promoted to *A*.ndim by pre-pending 1's to it. Thus for an *A* of shape (2, 3, 4, 5), a *reps* of (2, 2) is treated as (1, 1, 2, 2).

Note : Although tile may be used for broadcasting, it is strongly recommended to use Mars' broadcasting operations and functions.

A [array_like] The input tensor.

reps [array_like] The number of repetitions of *A* along each axis.

c [Tensor] The tiled output tensor.

repeat : Repeat elements of a tensor. broadcast_to : Broadcast a tensor to a new shape

```
>>> import mars.tensor as mt
```

```
>>> a = mt.array([0, 1, 2])
>>> mt.tile(a, 2).execute()
array([0, 1, 2, 0, 1, 2])
>>> mt.tile(a, (2, 2)).execute()
array([[0, 1, 2, 0, 1, 2],
       [0, 1, 2, 0, 1, 2]])
>>> mt.tile(a, (2, 1, 2)).execute()
array([[[0, 1, 2, 0, 1, 2]],
       [[0, 1, 2, 0, 1, 2]]])
```

```
>>> b = mt.array([[1, 2], [3, 4]])
>>> mt.tile(b, 2).execute()
array([[1, 2, 1, 2],
       [3, 4, 3, 4]])
>>> mt.tile(b, (2, 1)).execute()
array([[1, 2],
       [3, 4],
       [1, 2],
       [3, 4]])
```

```
>>> c = mt.array([1,2,3,4])
>>> mt.tile(c,(4,1)).execute()
array([[1, 2, 3, 4],
       [1, 2, 3, 4],
       [1, 2, 3, 4],
       [1, 2, 3, 4]])
```

mars.tensor.repeat

`mars.tensor.`**`repeat`**(*a*, *repeats*, *axis=None*)

Repeat elements of a tensor.

a [array_like] Input tensor.

repeats [int or tensor of ints] The number of repetitions for each element. *repeats* is broadcasted to fit the shape of the given axis.

axis [int, optional] The axis along which to repeat values. By default, use the flattened input tensor, and return a flat output tensor.

repeated_tensor [Tensor] Output array which has the same shape as *a*, except along the given axis.

tile : Tile a tensor.

```
>>> import mars.tensor as mt
```

```
>>> mt.repeat(3, 4).execute()
array([3, 3, 3, 3])
>>> x = mt.array([[1,2],[3,4]])
>>> mt.repeat(x, 2).execute()
array([1, 1, 2, 2, 3, 3, 4, 4])
>>> mt.repeat(x, 3, axis=1).execute()
array([[1, 1, 1, 2, 2, 2],
       [3, 3, 3, 4, 4, 4]])
>>> mt.repeat(x, [1, 2], axis=0).execute()
array([[1, 2],
       [3, 4],
       [3, 4]])
```

Rearranging elements

mars.tensor.flip	Reverse the order of elements in a tensor along the given axis.
mars.tensor.fliplr	Flip tensor in the left/right direction.
mars.tensor.flipud	Flip tensor in the up/down direction.
mars.tensor.reshape	Gives a new shape to a tensor without changing its data.
mars.tensor.roll	Roll tensor elements along a given axis.

mars.tensor.flip

mars.tensor.**flip**(*m*, *axis*)

Reverse the order of elements in a tensor along the given axis.

The shape of the array is preserved, but the elements are reordered.

m [array_like] Input tensor.

axis [integer] Axis in tensor, which entries are reversed.

out [array_like] A view of *m* with the entries of axis reversed. Since a view is returned, this operation is done in constant time.

flipud : Flip a tensor vertically (axis=0). fliplr : Flip a tensor horizontally (axis=1).

flip(m, 0) is equivalent to flipud(m). flip(m, 1) is equivalent to fliplr(m). flip(m, n) corresponds to `m[..., ::-1, ...]` with `::-1` at position n.

```
>>> import mars.tensor as mt
```

```
>>> A = mt.arange(8).reshape((2,2,2))
>>> A.execute()
array([[[0, 1],
        [2, 3]],
```

[[4, 5], [6, 7]]])

```
>>> mt.flip(A, 0).execute()
array([[[4, 5],
        [6, 7]],
```

[[0, 1], [2, 3]]])

```
>>> mt.flip(A, 1).execute()
array([[[2, 3],
        [0, 1]],
```

[[6, 7], [4, 5]]])

```
>>> A = mt.random.randn(3,4,5)
>>> mt.all(mt.flip(A,2) == A[:,:,::-1,...]).execute()
True
```

mars.tensor.fliplr

mars.tensor.**fliplr**(*m*)

Flip tensor in the left/right direction.

Flip the entries in each row in the left/right direction. Columns are preserved, but appear in a different order than before.

m [array_like] Input tensor, must be at least 2-D.

f [Tensor] A view of *m* with the columns reversed. Since a view is returned, this operation is $\mathcal{O}(1)$.

flipud : Flip array in the up/down direction. rot90 : Rotate array counterclockwise.

Equivalent to m[:,::-1]. Requires the tensor to be at least 2-D.

```
>>> import mars.tensor as mt
```

```
>>> A = mt.diag([1.,2.,3.])
>>> A.execute()
array([[ 1.,  0.,  0.],
       [ 0.,  2.,  0.],
       [ 0.,  0.,  3.]])
>>> mt.fliplr(A).execute()
array([[ 0.,  0.,  1.],
       [ 0.,  2.,  0.],
       [ 3.,  0.,  0.]])
```

```
>>> A = mt.random.randn(2,3,5)
>>> mt.all(mt.fliplr(A) == A[:,::-1,...]).execute()
True
```

mars.tensor.flipud

mars.tensor.**flipud**(*m*)

Flip tensor in the up/down direction.

Flip the entries in each column in the up/down direction. Rows are preserved, but appear in a different order than before.

m [array_like] Input tensor.

out [array_like] A view of *m* with the rows reversed. Since a view is returned, this operation is $\mathcal{O}(1)$.

fliplr : Flip tensor in the left/right direction. rot90 : Rotate tensor counterclockwise.

Equivalent to `m[::-1,...]`. Does not require the tensor to be two-dimensional.

```
>>> import mars.tensor as mt
```

```
>>> A = mt.diag([1.0, 2, 3])
>>> A.execute()
array([[ 1.,  0.,  0.],
       [ 0.,  2.,  0.],
       [ 0.,  0.,  3.]])
>>> mt.flipud(A).execute()
array([[ 0.,  0.,  3.],
       [ 0.,  2.,  0.],
       [ 1.,  0.,  0.]])
```

```
>>> A = mt.random.randn(2,3,5)
>>> mt.all(mt.flipud(A) == A[::-1,...]).execute()
True
```

```
>>> mt.flipud([1,2]).execute()
array([2, 1])
```

mars.tensor.roll

`mars.tensor.`**`roll`**(*a*, *shift*, *axis=None*)

Roll tensor elements along a given axis.

Elements that roll beyond the last position are re-introduced at the first.

a [array_like] Input tensor.

shift [int or tuple of ints] The number of places by which elements are shifted. If a tuple, then *axis* must be a tuple of the same size, and each of the given axes is shifted by the corresponding number. If an int while *axis* is a tuple of ints, then the same value is used for all given axes.

axis [int or tuple of ints, optional] Axis or axes along which elements are shifted. By default, the tensor is flattened before shifting, after which the original shape is restored.

res [Tensor] Output tensor, with the same shape as *a*.

rollaxis [Roll the specified axis backwards, until it lies in a] given position.

Supports rolling over multiple dimensions simultaneously.

```
>>> import mars.tensor as mt
```

```
>>> x = mt.arange(10)
>>> mt.roll(x, 2).execute()
array([8, 9, 0, 1, 2, 3, 4, 5, 6, 7])
```

```
>>> x2 = mt.reshape(x, (2,5))
>>> x2.execute()
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> mt.roll(x2, 1).execute()
array([[9, 0, 1, 2, 3],
       [4, 5, 6, 7, 8]])
>>> mt.roll(x2, 1, axis=0).execute()
array([[5, 6, 7, 8, 9],
       [0, 1, 2, 3, 4]])
>>> mt.roll(x2, 1, axis=1).execute()
array([[4, 0, 1, 2, 3],
       [9, 5, 6, 7, 8]])
```

2.6.3 Binary Operations

Elementwise bit operations

mars.tensor.bitwise_and	Compute the bit-wise AND of two tensors element-wise.
mars.tensor.bitwise_or	Compute the bit-wise OR of two tensors element-wise.
mars.tensor.bitwise_xor	Compute the bit-wise XOR of two arrays element-wise.
mars.tensor.invert	Compute bit-wise inversion, or bit-wise NOT, element-wise.
mars.tensor.left_shift	Shift the bits of an integer to the left.
mars.tensor.right_shift	Shift the bits of an integer to the right.

2.6.4 Discrete Fourier Transform

Standard FFTs

mars.tensor.fft.fft	Compute the one-dimensional discrete Fourier Transform.
mars.tensor.fft.ifft	Compute the one-dimensional inverse discrete Fourier Transform.
mars.tensor.fft.fft2	Compute the 2-dimensional discrete Fourier Transform
mars.tensor.fft.ifft2	Compute the 2-dimensional inverse discrete Fourier Transform.
mars.tensor.fft.fftn	Compute the N-dimensional discrete Fourier Transform.
mars.tensor.fft.ifftn	Compute the N-dimensional inverse discrete Fourier Transform.

mars.tensor.fft.fft

mars.tensor.fft.**fft**(*a*, *n=None*, *axis=-1*, *norm=None*)

Compute the one-dimensional discrete Fourier Transform.

This function computes the one-dimensional *n*-point discrete Fourier Transform (DFT) with the efficient Fast Fourier Transform (FFT) algorithm [CT].

a [array_like] Input tensor, can be complex.

n [int, optional] Length of the transformed axis of the output. If *n* is smaller than the length of the input, the input is cropped. If it is larger, the input is padded with zeros. If *n* is not given, the length of the input along the axis specified by *axis* is used.

axis [int, optional] Axis over which to compute the FFT. If not given, the last axis is used.

norm [{None, "ortho"}, optional] Normalization mode (see *mt.fft*). Default is None.

out [complex Tensor] The truncated or zero-padded input, transformed along the axis indicated by *axis*, or the last one if *axis* is not specified.

IndexError if *axes* is larger than the last axis of *a*.

mt.fft : for definition of the DFT and conventions used. ifft : The inverse of *fft*. fft2 : The two-dimensional FFT. fftn : The *n*-dimensional FFT. rfftn : The *n*-dimensional FFT of real input. fftfreq : Frequency bins for given FFT parameters.

FFT (Fast Fourier Transform) refers to a way the discrete Fourier Transform (DFT) can be calculated efficiently, by using symmetries in the calculated terms. The symmetry is highest when *n* is a power of 2, and the transform is therefore most efficient for these sizes.

The DFT is defined, with the conventions used in this implementation, in the documentation for the *numpy.fft* module.

```
>>> import mars.tensor as mt
```

```
>>> mt.fft.fft(mt.exp(2j * mt.pi * mt.arange(8) / 8)).execute()
array([-2.33486982e-16+1.14423775e-17j,  8.00000000e+00-6.89018570e-16j,
        2.33486982e-16+2.33486982e-16j,  0.00000000e+00+0.00000000e+00j,
       -1.14423775e-17+2.33486982e-16j,  0.00000000e+00+1.99159850e-16j,
        1.14423775e-17+1.14423775e-17j,  0.00000000e+00+0.00000000e+00j])
```

In this example, real input has an FFT which is Hermitian, i.e., symmetric in the real part and anti-symmetric in the imaginary part, as described in the *numpy.fft* documentation:

```
>>> import matplotlib.pyplot as plt
>>> t = mt.arange(256)
>>> sp = mt.fft.fft(mt.sin(t))
>>> freq = mt.fft.fftfreq(t.shape[-1])
>>> plt.plot(freq.execute(), sp.real.execute(), freq.execute(), sp.imag.execute())
[<matplotlib.lines.Line2D object at 0x...>, <matplotlib.lines.Line2D object at 0x.
↪..>]
>>> plt.show()
```

mars.tensor.fft.ifft

mars.tensor.fft.**ifft**(*a*, *n=None*, *axis=-1*, *norm=None*)

Compute the one-dimensional inverse discrete Fourier Transform.

This function computes the inverse of the one-dimensional *n*-point discrete Fourier transform computed by *fft*. In other words, `ifft(fft(a)) == a` to within numerical accuracy. For a general description of the algorithm and definitions, see *mt.fft*.

The input should be ordered in the same way as is returned by *fft*, i.e.,

- `a[0]` should contain the zero frequency term,
- `a[1:n//2]` should contain the positive-frequency terms,
- `a[n//2 + 1:]` should contain the negative-frequency terms, in increasing order starting from the most negative frequency.

For an even number of input points, `A[n//2]` represents the sum of the values at the positive and negative Nyquist frequencies, as the two are aliased together. See *numpy.fft* for details.

a [array_like] Input tensor, can be complex.

n [int, optional] Length of the transformed axis of the output. If *n* is smaller than the length of the input, the input is cropped. If it is larger, the input is padded with zeros. If *n* is not given, the length of the input along the axis specified by *axis* is used. See notes about padding issues.

axis [int, optional] Axis over which to compute the inverse DFT. If not given, the last axis is used.

norm [{None, "ortho"}, optional] Normalization mode (see *numpy.fft*). Default is None.

out [complex Tensor] The truncated or zero-padded input, transformed along the axis indicated by *axis*, or the last one if *axis* is not specified.

IndexError If *axes* is larger than the last axis of *a*.

mt.fft : An introduction, with definitions and general explanations. fft : The one-dimensional (forward) FFT, of which *ifft* is the inverse ifft2 : The two-dimensional inverse FFT. ifftn : The n-dimensional inverse FFT.

If the input parameter *n* is larger than the size of the input, the input is padded by appending zeros at the end. Even though this is the common approach, it might lead to surprising results. If a different padding is desired, it must be performed before calling *ifft*.

```
>>> import mars.tensor as mt
```

```
>>> mt.fft.ifft([0, 4, 0, 0]).execute()
array([ 1.+0.j,  0.+1.j, -1.+0.j,  0.-1.j])
```

Create and plot a band-limited signal with random phases:

```
>>> import matplotlib.pyplot as plt
>>> t = mt.arange(400)
>>> n = mt.zeros((400,), dtype=complex)
>>> n[40:60] = mt.exp(1j*mt.random.uniform(0, 2*mt.pi, (20,)))
>>> s = mt.fft.ifft(n)
>>> plt.plot(t.execute(), s.real.execute(), 'b-', t.execute(), s.imag.execute(),
↪'r--')
...
>>> plt.legend(('real', 'imaginary'))
```

(continues on next page)

(continued from previous page)

```
...
>>> plt.show()
```

mars.tensor.fft.fft2

mars.tensor.fft.**fft2**(*a*, *s=None*, *axes=(-2, -1)*, *norm=None*)

Compute the 2-dimensional discrete Fourier Transform

This function computes the *n*-dimensional discrete Fourier Transform over any axes in an *M*-dimensional array by means of the Fast Fourier Transform (FFT). By default, the transform is computed over the last two axes of the input array, i.e., a 2-dimensional FFT.

a [array_like] Input tensor, can be complex

s [sequence of ints, optional] Shape (length of each transformed axis) of the output (`s[0]` refers to axis 0, `s[1]` to axis 1, etc.). This corresponds to `n` for `fft(x, n)`. Along each axis, if the given shape is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. if *s* is not given, the shape of the input along the axes specified by *axes* is used.

axes [sequence of ints, optional] Axes over which to compute the FFT. If not given, the last two axes are used. A repeated index in *axes* means the transform over that axis is performed multiple times. A one-element sequence means that a one-dimensional FFT is performed.

norm [{None, "ortho"}, optional] Normalization mode (see *mt.fft*). Default is None.

out [complex Tensor] The truncated or zero-padded input, transformed along the axes indicated by *axes*, or the last two axes if *axes* is not given.

ValueError If *s* and *axes* have different length, or *axes* not given and `len(s) != 2`.

IndexError If an element of *axes* is larger than than the number of axes of *a*.

mt.fft [Overall view of discrete Fourier transforms, with definitions] and conventions used.

ifft2 : The inverse two-dimensional FFT. fft : The one-dimensional FFT. fftn : The *n*-dimensional FFT. fftshift : Shifts zero-frequency terms to the center of the array.

For two-dimensional input, swaps first and third quadrants, and second and fourth quadrants.

fft2 is just *fftn* with a different default for *axes*.

The output, analogously to *fft*, contains the term for zero frequency in the low-order corner of the transformed axes, the positive frequency terms in the first half of these axes, the term for the Nyquist frequency in the middle of the axes and the negative frequency terms in the second half of the axes, in order of decreasingly negative frequency.

See *fftn* for details and a plotting example, and *mt.fft* for definitions and conventions used.

```
>>> import mars.tensor as mt
```

```
>>> a = mt.mgrid[:5, :5][0]
>>> mt.fft.fft2(a).execute()
array([[ 50.0 +0.j        ,   0.0 +0.j        ,   0.0 +0.j        ,
          0.0 +0.j        ,   0.0 +0.j        ],
       [-12.5+17.20477401j,   0.0 +0.j        ,   0.0 +0.j        ,
          0.0 +0.j        ,   0.0 +0.j        ],
```

(continues on next page)

(continued from previous page)

```
       [-12.5 +4.0614962j ,   0.0 +0.j        ,   0.0 +0.j        ,
          0.0 +0.j        ,   0.0 +0.j        ],
       [-12.5 -4.0614962j ,   0.0 +0.j        ,   0.0 +0.j        ,
            0.0 +0.j        ,   0.0 +0.j        ],
       [-12.5-17.20477401j,   0.0 +0.j        ,   0.0 +0.j        ,
          0.0 +0.j        ,   0.0 +0.j        ]])
```

mars.tensor.fft.ifft2

mars.tensor.fft.**ifft2**(*a*, *s=None*, *axes=(-2*, *-1)*, *norm=None*)

Compute the 2-dimensional inverse discrete Fourier Transform.

This function computes the inverse of the 2-dimensional discrete Fourier Transform over any number of axes in an M-dimensional array by means of the Fast Fourier Transform (FFT). In other words, `ifft2(fft2(a)) == a` to within numerical accuracy. By default, the inverse transform is computed over the last two axes of the input array.

The input, analogously to *ifft*, should be ordered in the same way as is returned by *fft2*, i.e. it should have the term for zero frequency in the low-order corner of the two axes, the positive frequency terms in the first half of these axes, the term for the Nyquist frequency in the middle of the axes and the negative frequency terms in the second half of both axes, in order of decreasingly negative frequency.

a [array_like] Input tensor, can be complex.

s [sequence of ints, optional] Shape (length of each axis) of the output (`s[0]` refers to axis 0, `s[1]` to axis 1, etc.). This corresponds to *n* for `ifft(x, n)`. Along each axis, if the given shape is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. if *s* is not given, the shape of the input along the axes specified by *axes* is used. See notes for issue on *ifft* zero padding.

axes [sequence of ints, optional] Axes over which to compute the FFT. If not given, the last two axes are used. A repeated index in *axes* means the transform over that axis is performed multiple times. A one-element sequence means that a one-dimensional FFT is performed.

norm [{None, “ortho”}, optional] Normalization mode (see *mt.fft*). Default is None.

out [complex Tensor] The truncated or zero-padded input, transformed along the axes indicated by *axes*, or the last two axes if *axes* is not given.

ValueError If *s* and *axes* have different length, or *axes* not given and `len(s) != 2`.

IndexError If an element of *axes* is larger than than the number of axes of *a*.

mt.fft [Overall view of discrete Fourier transforms, with definitions] and conventions used.

fft2 : The forward 2-dimensional FFT, of which *ifft2* is the inverse. ifftn : The inverse of the *n*-dimensional FFT. fft : The one-dimensional FFT. ifft : The one-dimensional inverse FFT.

ifft2 is just *ifftn* with a different default for *axes*.

See *ifftn* for details and a plotting example, and *numpy.fft* for definition and conventions used.

Zero-padding, analogously with *ifft*, is performed by appending zeros to the input along the specified dimension. Although this is the common approach, it might lead to surprising results. If another form of zero padding is desired, it must be performed before *ifft2* is called.

```
>>> import mars.tensor as mt
```

```
>>> a = 4 * mt.eye(4)
>>> mt.fft.ifft2(a).execute()
array([[ 1.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j,  0.+0.j,  1.+0.j],
       [ 0.+0.j,  0.+0.j,  1.+0.j,  0.+0.j],
       [ 0.+0.j,  1.+0.j,  0.+0.j,  0.+0.j]])
```

mars.tensor.fft.fftn

mars.tensor.fft.**fftn**(*a*, *s=None*, *axes=None*, *norm=None*)

Compute the N-dimensional discrete Fourier Transform.

This function computes the *N*-dimensional discrete Fourier Transform over any number of axes in an *M*-dimensional tensor by means of the Fast Fourier Transform (FFT).

a [array_like] Input tensor, can be complex.

s [sequence of ints, optional] Shape (length of each transformed axis) of the output (`s[0]` refers to axis 0, `s[1]` to axis 1, etc.). This corresponds to `n` for `fft(x, n)`. Along any axis, if the given shape is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. if *s* is not given, the shape of the input along the axes specified by *axes* is used.

axes [sequence of ints, optional] Axes over which to compute the FFT. If not given, the last `len(s)` axes are used, or all axes if *s* is also not specified. Repeated indices in *axes* means that the transform over that axis is performed multiple times.

norm [{None, "ortho"}, optional] Normalization mode (see *mt.fft*). Default is None.

out [complex Tensor] The truncated or zero-padded input, transformed along the axes indicated by *axes*, or by a combination of *s* and *a*, as explained in the parameters section above.

ValueError If *s* and *axes* have different length.

IndexError If an element of *axes* is larger than than the number of axes of *a*.

mt.fft [Overall view of discrete Fourier transforms, with definitions] and conventions used.

ifftn : The inverse of *fftn*, the inverse *n*-dimensional FFT. fft : The one-dimensional FFT, with definitions and conventions used. rfftn : The *n*-dimensional FFT of real input. fft2 : The two-dimensional FFT. fftshift : Shifts zero-frequency terms to centre of tensor

The output, analogously to *fft*, contains the term for zero frequency in the low-order corner of all axes, the positive frequency terms in the first half of all axes, the term for the Nyquist frequency in the middle of all axes and the negative frequency terms in the second half of all axes, in order of decreasingly negative frequency.

See *mt.fft* for details, definitions and conventions used.

```
>>> import mars.tensor as mt
```

```
>>> a = mt.mgrid[:3, :3, :3][0]
>>> mt.fft.fftn(a, axes=(1, 2)).execute()
array([[[  0.+0.j,   0.+0.j,   0.+0.j],
        [  0.+0.j,   0.+0.j,   0.+0.j],
        [  0.+0.j,   0.+0.j,   0.+0.j]],
       [[  9.+0.j,   0.+0.j,   0.+0.j],
        [  0.+0.j,   0.+0.j,   0.+0.j],
```

(continues on next page)

(continued from previous page)

```
        [  0.+0.j,   0.+0.j,   0.+0.j]],
       [[ 18.+0.j,   0.+0.j,   0.+0.j],
        [  0.+0.j,   0.+0.j,   0.+0.j],
        [  0.+0.j,   0.+0.j,   0.+0.j]]])
>>> mt.fft.fftn(a, (2, 2), axes=(0, 1)).execute()
array([[[ 2.+0.j,  2.+0.j,  2.+0.j],
        [ 0.+0.j,  0.+0.j,  0.+0.j]],
       [[-2.+0.j, -2.+0.j, -2.+0.j],
        [ 0.+0.j,  0.+0.j,  0.+0.j]]])
```

```
>>> import matplotlib.pyplot as plt
>>> [X, Y] = mt.meshgrid(2 * mt.pi * mt.arange(200) / 12,
...                      2 * mt.pi * mt.arange(200) / 34)
>>> S = mt.sin(X) + mt.cos(Y) + mt.random.uniform(0, 1, X.shape)
>>> FS = mt.fft.fftn(S)
>>> plt.imshow(mt.log(mt.abs(mt.fft.fftshift(FS))**2).execute())
<matplotlib.image.AxesImage object at 0x...>
>>> plt.show()
```

mars.tensor.fft.ifftn

mars.tensor.fft.**ifftn**(*a*, *s=None*, *axes=None*, *norm=None*)

Compute the N-dimensional inverse discrete Fourier Transform.

This function computes the inverse of the N-dimensional discrete Fourier Transform over any number of axes in an M-dimensional tensor by means of the Fast Fourier Transform (FFT). In other words, `ifftn(fftn(a)) == a` to within numerical accuracy. For a description of the definitions and conventions used, see *mt.fft*.

The input, analogously to *ifft*, should be ordered in the same way as is returned by *fftn*, i.e. it should have the term for zero frequency in all axes in the low-order corner, the positive frequency terms in the first half of all axes, the term for the Nyquist frequency in the middle of all axes and the negative frequency terms in the second half of all axes, in order of decreasingly negative frequency.

a [array_like] Input tensor, can be complex.

s [sequence of ints, optional] Shape (length of each transformed axis) of the output (`s[0]` refers to axis 0, `s[1]` to axis 1, etc.). This corresponds to `n` for `ifft(x, n)`. Along any axis, if the given shape is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. if *s* is not given, the shape of the input along the axes specified by *axes* is used. See notes for issue on *ifft* zero padding.

axes [sequence of ints, optional] Axes over which to compute the IFFT. If not given, the last `len(s)` axes are used, or all axes if *s* is also not specified. Repeated indices in *axes* means that the inverse transform over that axis is performed multiple times.

norm [{None, "ortho"}, optional] Normalization mode (see *mt.fft*). Default is None.

out [complex Tensor] The truncated or zero-padded input, transformed along the axes indicated by *axes*, or by a combination of *s* or *a*, as explained in the parameters section above.

ValueError If *s* and *axes* have different length.

IndexError If an element of *axes* is larger than than the number of axes of *a*.

mt.fft [Overall view of discrete Fourier transforms, with definitions] and conventions used.

fftn : The forward *n*-dimensional FFT, of which *ifftn* is the inverse. ifft : The one-dimensional inverse FFT. ifft2 : The two-dimensional inverse FFT. ifftshift : Undoes *fftshift*, shifts zero-frequency terms to beginning

of tensor.

See *mt.fft* for definitions and conventions used.

Zero-padding, analogously with *ifft*, is performed by appending zeros to the input along the specified dimension. Although this is the common approach, it might lead to surprising results. If another form of zero padding is desired, it must be performed before *ifftn* is called.

```
>>> import mars.tensor as mt
```

```
>>> a = mt.eye(4)
>>> mt.fft.ifftn(mt.fft.fftn(a, axes=(0,)), axes=(1,)).execute()
array([[ 1.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
       [ 0.+0.j,  1.+0.j,  0.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j,  1.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j,  0.+0.j,  1.+0.j]])
```

Create and plot an image with band-limited frequency content:

```
>>> import matplotlib.pyplot as plt
>>> n = mt.zeros((200,200), dtype=complex)
>>> n[60:80, 20:40] = mt.exp(1j*mt.random.uniform(0, 2*mt.pi, (20, 20)))
>>> im = mt.fft.ifftn(n).real
>>> plt.imshow(im.execute())
<matplotlib.image.AxesImage object at 0x...>
>>> plt.show()
```

Real FFTs

mars.tensor.fft.rfft	Compute the one-dimensional discrete Fourier Transform for real input.
mars.tensor.fft.irfft	Compute the inverse of the n-point DFT for real input.
mars.tensor.fft.rfft2	Compute the 2-dimensional FFT of a real tensor.
mars.tensor.fft.irfft2	Compute the 2-dimensional inverse FFT of a real array.
mars.tensor.fft.rfftn	Compute the N-dimensional discrete Fourier Transform for real input.
mars.tensor.fft.irfftn	Compute the inverse of the N-dimensional FFT of real input.

mars.tensor.fft.rfft

mars.tensor.fft.**rfft**(*a*, *n=None*, *axis=-1*, *norm=None*)

Compute the one-dimensional discrete Fourier Transform for real input.

This function computes the one-dimensional *n*-point discrete Fourier Transform (DFT) of a real-valued array by means of an efficient algorithm called the Fast Fourier Transform (FFT).

a [array_like] Input tensor

n [int, optional] Number of points along transformation axis in the input to use. If *n* is smaller than the length of the input, the input is cropped. If it is larger, the input is padded with zeros. If *n* is not given, the length of the input along the axis specified by *axis* is used.

axis [int, optional] Axis over which to compute the FFT. If not given, the last axis is used.

norm [{None, "ortho"}, optional] Normalization mode (see *mt.fft*). Default is None.

out [complex Tensor] The truncated or zero-padded input, transformed along the axis indicated by *axis*, or the last one if *axis* is not specified. If *n* is even, the length of the transformed axis is `(n/2)+1`. If *n* is odd, the length is `(n+1)/2`.

IndexError If *axis* is larger than the last axis of *a*.

mt.fft : For definition of the DFT and conventions used. irfft : The inverse of *rfft*. fft : The one-dimensional FFT of general (complex) input. fftn : The *n*-dimensional FFT. rfftn : The *n*-dimensional FFT of real input.

When the DFT is computed for purely real input, the output is Hermitian-symmetric, i.e. the negative frequency terms are just the complex conjugates of the corresponding positive-frequency terms, and the negative-frequency terms are therefore redundant. This function does not compute the negative frequency terms, and the length of the transformed axis of the output is therefore `n//2 + 1`.

When `A = rfft(a)` and fs is the sampling frequency, `A[0]` contains the zero-frequency term 0*fs, which is real due to Hermitian symmetry.

If *n* is even, `A[-1]` contains the term representing both positive and negative Nyquist frequency (+fs/2 and -fs/2), and must also be purely real. If *n* is odd, there is no term at fs/2; `A[-1]` contains the largest positive frequency (fs/2*(n-1)/n), and is complex in the general case.

If the input *a* contains an imaginary part, it is silently discarded.

```
>>> import mars.tensor as mt
```

```
>>> mt.fft.fft([0, 1, 0, 0]).execute()
array([ 1.+0.j,  0.-1.j, -1.+0.j,  0.+1.j])
>>> mt.fft.rfft([0, 1, 0, 0]).execute()
array([ 1.+0.j,  0.-1.j, -1.+0.j])
```

Notice how the final element of the *fft* output is the complex conjugate of the second element, for real input. For *rfft*, this symmetry is exploited to compute only the non-negative frequency terms.

mars.tensor.fft.irfft

`mars.tensor.fft.`**`irfft`**(*a*, *n=None*, *axis=-1*, *norm=None*)

Compute the inverse of the n-point DFT for real input.

This function computes the inverse of the one-dimensional *n*-point discrete Fourier Transform of real input computed by *rfft*. In other words, `irfft(rfft(a), len(a)) == a` to within numerical accuracy. (See Notes below for why `len(a)` is necessary here.)

The input is expected to be in the form returned by *rfft*, i.e. the real zero-frequency term followed by the complex positive frequency terms in order of increasing frequency. Since the discrete Fourier Transform of real input is Hermitian-symmetric, the negative frequency terms are taken to be the complex conjugates of the corresponding positive frequency terms.

a [array_like] The input tensor.

n [int, optional] Length of the transformed axis of the output. For *n* output points, `n//2+1` input points are necessary. If the input is longer than this, it is cropped. If it is shorter than this, it is padded with zeros. If *n* is not given, it is determined from the length of the input along the axis specified by *axis*.

axis [int, optional] Axis over which to compute the inverse FFT. If not given, the last axis is used.

norm [{None, "ortho"}, optional] Normalization mode (see *mt.fft*). Default is None.

out [Tensor] The truncated or zero-padded input, transformed along the axis indicated by *axis*, or the last one if *axis* is not specified. The length of the transformed axis is *n*, or, if *n* is not given, `2*(m-1)` where `m` is the length of the transformed axis of the input. To get an odd number of output points, *n* must be specified.

IndexError If *axis* is larger than the last axis of *a*.

mt.fft : For definition of the DFT and conventions used. rfft : The one-dimensional FFT of real input, of which *irfft* is inverse. fft : The one-dimensional FFT. irfft2 : The inverse of the two-dimensional FFT of real input. irfftn : The inverse of the *n*-dimensional FFT of real input.

Returns the real valued *n*-point inverse discrete Fourier transform of *a*, where *a* contains the non-negative frequency terms of a Hermitian-symmetric sequence. *n* is the length of the result, not the input.

If you specify an *n* such that *a* must be zero-padded or truncated, the extra/removed values will be added/removed at high frequencies. One can thus resample a series to *m* points via Fourier interpolation by: `a_resamp = irfft(rfft(a), m)`.

```
>>> import mars.tenosr as mt
```

```
>>> mt.fft.ifft([1, -1j, -1, 1j]).execute()
array([ 0.+0.j,  1.+0.j,  0.+0.j,  0.+0.j])
>>> mt.fft.irfft([1, -1j, -1]).execute()
array([ 0.,  1.,  0.,  0.])
```

Notice how the last term in the input to the ordinary *ifft* is the complex conjugate of the second term, and the output has zero imaginary part everywhere. When calling *irfft*, the negative frequencies are not specified, and the output array is purely real.

mars.tensor.fft.rfft2

mars.tensor.fft.**rfft2**(*a*, *s=None*, *axes=(-2*, *-1)*, *norm=None*)

Compute the 2-dimensional FFT of a real tensor.

a [array_like] Input tensor, taken to be real.

s [sequence of ints, optional] Shape of the FFT.

axes [sequence of ints, optional] Axes over which to compute the FFT.

norm [{None, "ortho"}, optional] Normalization mode (see *mt.fft*). Default is None.

out [Tensor] The result of the real 2-D FFT.

rfftn [Compute the N-dimensional discrete Fourier Transform for real] input.

This is really just *rfftn* with different default behavior. For more details see *rfftn*.

mars.tensor.fft.irfft2

mars.tensor.fft.**irfft2**(*a*, *s=None*, *axes=(-2*, *-1)*, *norm=None*)

Compute the 2-dimensional inverse FFT of a real array.

a [array_like] The input tensor

s [sequence of ints, optional] Shape of the inverse FFT.

axes [sequence of ints, optional] The axes over which to compute the inverse fft. Default is the last two axes.

norm [{None, "ortho"}, optional] Normalization mode (see *mt.fft*). Default is None.

out [Tensor] The result of the inverse real 2-D FFT.

irfftn : Compute the inverse of the N-dimensional FFT of real input.

This is really *irfftn* with different defaults. For more details see *irfftn*.

mars.tensor.fft.rfftn

mars.tensor.fft.**rfftn**(*a*, *s=None*, *axes=None*, *norm=None*)

Compute the N-dimensional discrete Fourier Transform for real input.

This function computes the N-dimensional discrete Fourier Transform over any number of axes in an M-dimensional real tensor by means of the Fast Fourier Transform (FFT). By default, all axes are transformed, with the real transform performed over the last axis, while the remaining transforms are complex.

a [array_like] Input tensor, taken to be real.

s [sequence of ints, optional] Shape (length along each transformed axis) to use from the input. (`s[0]` refers to axis 0, `s[1]` to axis 1, etc.). The final element of *s* corresponds to *n* for `rfft(x, n)`, while for the remaining axes, it corresponds to *n* for `fft(x, n)`. Along any axis, if the given shape is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. if *s* is not given, the shape of the input along the axes specified by *axes* is used.

axes [sequence of ints, optional] Axes over which to compute the FFT. If not given, the last `len(s)` axes are used, or all axes if *s* is also not specified.

norm [{None, "ortho"}, optional] Normalization mode (see *mt.fft*). Default is None.

out [complex Tensor] The truncated or zero-padded input, transformed along the axes indicated by *axes*, or by a combination of *s* and *a*, as explained in the parameters section above. The length of the last axis transformed will be `s[-1]//2+1`, while the remaining transformed axes will have lengths according to *s*, or unchanged from the input.

ValueError If *s* and *axes* have different length.

IndexError If an element of *axes* is larger than than the number of axes of *a*.

irfftn [The inverse of *rfftn*, i.e. the inverse of the n-dimensional FFT] of real input.

fft : The one-dimensional FFT, with definitions and conventions used. rfft : The one-dimensional FFT of real input. fftn : The n-dimensional FFT. rfft2 : The two-dimensional FFT of real input.

The transform for real input is performed over the last transformation axis, as by *rfft*, then the transform over the remaining axes is performed as by *fftn*. The order of the output is as for *rfft* for the final transformation axis, and as for *fftn* for the remaining transformation axes.

See *fft* for details, definitions and conventions used.

```
>>> import mars.tensor as mt
```

```
>>> a = mt.ones((2, 2, 2))
>>> mt.fft.rfftn(a).execute()
array([[[ 8.+0.j,  0.+0.j],
        [ 0.+0.j,  0.+0.j]],
       [[ 0.+0.j,  0.+0.j],
        [ 0.+0.j,  0.+0.j]]])
```

```
>>> mt.fft.rfftn(a, axes=(2, 0)).execute()
array([[[ 4.+0.j,  0.+0.j],
        [ 4.+0.j,  0.+0.j]],
       [[ 0.+0.j,  0.+0.j],
        [ 0.+0.j,  0.+0.j]]])
```

mars.tensor.fft.irfftn

mars.tensor.fft.**irfftn**(*a*, *s=None*, *axes=None*, *norm=None*)

Compute the inverse of the N-dimensional FFT of real input.

This function computes the inverse of the N-dimensional discrete Fourier Transform for real input over any number of axes in an M-dimensional tensor by means of the Fast Fourier Transform (FFT). In other words, `irfftn(rfftn(a), a.shape) == a` to within numerical accuracy. (The `a.shape` is necessary like `len(a)` is for *irfft*, and for the same reason.)

The input should be ordered in the same way as is returned by *rfftn*, i.e. as for *irfft* for the final transformation axis, and as for *ifftn* along all the other axes.

a [array_like] Input tensor.

s [sequence of ints, optional] Shape (length of each transformed axis) of the output (`s[0]` refers to axis 0, `s[1]` to axis 1, etc.). *s* is also the number of input points used along this axis, except for the last axis, where `s[-1]//2+1` points of the input are used. Along any axis, if the shape indicated by *s* is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. If *s* is not given, the shape of the input along the axes specified by *axes* is used.

axes [sequence of ints, optional] Axes over which to compute the inverse FFT. If not given, the last *len(s)* axes are used, or all axes if *s* is also not specified. Repeated indices in *axes* means that the inverse transform over that axis is performed multiple times.

norm [{None, “ortho”}, optional] Normalization mode (see *mt.fft*). Default is None.

out [Tensor] The truncated or zero-padded input, transformed along the axes indicated by *axes*, or by a combination of *s* or *a*, as explained in the parameters section above. The length of each transformed axis is as given by the corresponding element of *s*, or the length of the input in every axis except for the last one if *s* is not given. In the final transformed axis the length of the output when *s* is not given is `2*(m-1)` where `m` is the length of the final transformed axis of the input. To get an odd number of output points in the final axis, *s* must be specified.

ValueError If *s* and *axes* have different length.

IndexError If an element of *axes* is larger than than the number of axes of *a*.

rfftn [The forward n-dimensional FFT of real input,] of which *ifftn* is the inverse.

fft : The one-dimensional FFT, with definitions and conventions used. irfft : The inverse of the one-dimensional FFT of real input. irfft2 : The inverse of the two-dimensional FFT of real input.

See *fft* for definitions and conventions used.

See *rfft* for definitions and conventions used for real input.

```
>>> import mars.tensor as mt
```

```
>>> a = mt.zeros((3, 2, 2))
>>> a[0, 0, 0] = 3 * 2 * 2
>>> mt.fft.irfftn(a).execute()
array([[[ 1.,  1.],
        [ 1.,  1.]],
       [[ 1.,  1.],
        [ 1.,  1.]],
       [[ 1.,  1.],
        [ 1.,  1.]]])
```

Hermitian FFTs

mars.tensor.fft.hfft	Compute the FFT of a signal that has Hermitian symmetry, i.e., a real spectrum.
mars.tensor.fft.ihfft	Compute the inverse FFT of a signal that has Hermitian symmetry.

mars.tensor.fft.hfft

mars.tensor.fft.**hfft**(*a*, *n=None*, *axis=-1*, *norm=None*)

Compute the FFT of a signal that has Hermitian symmetry, i.e., a real spectrum.

a [array_like] The input tensor.

n [int, optional] Length of the transformed axis of the output. For *n* output points, `n//2 + 1` input points are necessary. If the input is longer than this, it is cropped. If it is shorter than this, it is padded with zeros. If *n* is not given, it is determined from the length of the input along the axis specified by *axis*.

axis [int, optional] Axis over which to compute the FFT. If not given, the last axis is used.

norm [{None, "ortho"}, optional] Normalization mode (see *mt.fft*). Default is None.

out [Tensor] The truncated or zero-padded input, transformed along the axis indicated by *axis*, or the last one if *axis* is not specified. The length of the transformed axis is *n*, or, if *n* is not given, `2*m - 2` where `m` is the length of the transformed axis of the input. To get an odd number of output points, *n* must be specified, for instance as `2*m - 1` in the typical case,

IndexError If *axis* is larger than the last axis of *a*.

rfft : Compute the one-dimensional FFT for real input. ihfft : The inverse of *hfft*.

hfft/*ihfft* are a pair analogous to *rfft*/*irfft*, but for the opposite case: here the signal has Hermitian symmetry in the time domain and is real in the frequency domain. So here it's *hfft* for which you must supply the length of the result if it is to be odd.

- even: `ihfft(hfft(a, 2*len(a) - 2) == a`, within roundoff error,
- odd: `ihfft(hfft(a, 2*len(a) - 1) == a`, within roundoff error.

```
>>> import mars.tensor as mt
```

```
>>> signal = mt.array([1, 2, 3, 4, 3, 2])
>>> mt.fft.fft(signal).execute()
array([ 15.+0.j,  -4.+0.j,   0.+0.j,  -1.-0.j,   0.+0.j,  -4.+0.j])
>>> mt.fft.hfft(signal[:4]).execute() # Input first half of signal
array([ 15.,  -4.,   0.,  -1.,   0.,  -4.])
>>> mt.fft.hfft(signal, 6).execute()  # Input entire signal and truncate
array([ 15.,  -4.,   0.,  -1.,   0.,  -4.])
```

```
>>> signal = mt.array([[1, 1.j], [-1.j, 2]])
>>> (mt.conj(signal.T) - signal).execute()    # check Hermitian symmetry
array([[ 0.-0.j,  0.+0.j],
       [ 0.+0.j,  0.-0.j]])
>>> freq_spectrum = mt.fft.hfft(signal)
>>> freq_spectrum.execute()
array([[ 1.,  1.],
       [ 2., -2.]])
```

mars.tensor.fft.ihfft

`mars.tensor.fft.`**`ihfft`**(*a*, *n=None*, *axis=-1*, *norm=None*)

Compute the inverse FFT of a signal that has Hermitian symmetry.

a [array_like] Input tensor.

n [int, optional] Length of the inverse FFT, the number of points along transformation axis in the input to use. If *n* is smaller than the length of the input, the input is cropped. If it is larger, the input is padded with zeros. If *n* is not given, the length of the input along the axis specified by *axis* is used.

axis [int, optional] Axis over which to compute the inverse FFT. If not given, the last axis is used.

norm [{None, "ortho"}, optional] Normalization mode (see *numpy.fft*). Default is None.

out [complex Tensor] The truncated or zero-padded input, transformed along the axis indicated by *axis*, or the last one if *axis* is not specified. The length of the transformed axis is `n//2 + 1`.

hfft, irfft

hfft/*ihfft* are a pair analogous to *rfft*/*irfft*, but for the opposite case: here the signal has Hermitian symmetry in the time domain and is real in the frequency domain. So here it's *hfft* for which you must supply the length of the result if it is to be odd:

- even: `ihfft(hfft(a, 2*len(a) - 2) == a`, within roundoff error,
- odd: `ihfft(hfft(a, 2*len(a) - 1) == a`, within roundoff error.

```
>>> import mars.tensor as mt
```

```
>>> spectrum = mt.array([ 15, -4, 0, -1, 0, -4])
>>> mt.fft.ifft(spectrum).execute()
array([ 1.+0.j,  2.-0.j,  3.+0.j,  4.+0.j,  3.+0.j,  2.-0.j])
>>> mt.fft.ihfft(spectrum).execute()
array([ 1.-0.j,  2.-0.j,  3.-0.j,  4.-0.j])
```

Helper routines

mars.tensor.fft.fftfreq	Return the Discrete Fourier Transform sample frequencies.
mars.tensor.fft.rfftfreq	Return the Discrete Fourier Transform sample frequencies (for usage with rfft, irfft).
mars.tensor.fft.fftshift	Shift the zero-frequency component to the center of the spectrum.
mars.tensor.fft.ifftshift	The inverse of *fftshift*.

mars.tensor.fft.fftfreq

mars.tensor.fft.**fftfreq**(*n*, *d=1.0*, *gpu=False*, *chunk_size=None*)

Return the Discrete Fourier Transform sample frequencies.

The returned float tensor *f* contains the frequency bin centers in cycles per unit of the sample spacing (with zero at the start). For instance, if the sample spacing is in seconds, then the frequency unit is cycles/second.

Given a window length *n* and a sample spacing *d*:

```
f = [0, 1, ...,   n/2-1,     -n/2, ..., -1] / (d*n)   if n is even
f = [0, 1, ..., (n-1)/2, -(n-1)/2, ..., -1] / (d*n)   if n is odd
```

n [int] Window length.

d [scalar, optional] Sample spacing (inverse of the sampling rate). Defaults to 1.

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

f [Tensor] Array of length *n* containing the sample frequencies.

```
>>> import mars.tensor as mt
```

```
>>> signal = mt.array([-2, 8, 6, 4, 1, 0, 3, 5], dtype=float)
>>> fourier = mt.fft.fft(signal)
>>> n = signal.size
>>> timestep = 0.1
>>> freq = mt.fft.fftfreq(n, d=timestep)
>>> freq.execute()
array([ 0.  ,  1.25,  2.5 ,  3.75, -5.  , -3.75, -2.5 , -1.25])
```

mars.tensor.fft.rfftfreq

mars.tensor.fft.**rfftfreq**(*n*, *d=1.0*, *gpu=False*, *chunk_size=None*)

Return the Discrete Fourier Transform sample frequencies (for usage with rfft, irfft).

The returned float tensor *f* contains the frequency bin centers in cycles per unit of the sample spacing (with zero at the start). For instance, if the sample spacing is in seconds, then the frequency unit is cycles/second.

Given a window length *n* and a sample spacing *d*:

```
f = [0, 1, ...,     n/2-1,     n/2] / (d*n)   if n is even
f = [0, 1, ..., (n-1)/2-1, (n-1)/2] / (d*n)   if n is odd
```

Unlike *fftfreq* (but like *scipy.fftpack.rfftfreq*) the Nyquist frequency component is considered to be positive.

n [int] Window length.

d [scalar, optional] Sample spacing (inverse of the sampling rate). Defaults to 1.

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

f [Tensor] Tensor of length `n//2 + 1` containing the sample frequencies.

```
>>> import mars.tensor as mt
```

```
>>> signal = mt.array([-2, 8, 6, 4, 1, 0, 3, 5, -3, 4], dtype=float)
>>> fourier = mt.fft.rfft(signal)
>>> n = signal.size
>>> sample_rate = 100
>>> freq = mt.fft.fftfreq(n, d=1./sample_rate)
>>> freq.execute()
array([  0.,  10.,  20.,  30.,  40., -50., -40., -30., -20., -10.])
>>> freq = mt.fft.rfftfreq(n, d=1./sample_rate)
>>> freq.execute()
array([  0.,  10.,  20.,  30.,  40.,  50.])
```

mars.tensor.fft.fftshift

mars.tensor.fft.**fftshift**(*x*, *axes=None*)

Shift the zero-frequency component to the center of the spectrum.

This function swaps half-spaces for all axes listed (defaults to all). Note that `y[0]` is the Nyquist component only if `len(x)` is even.

x [array_like] Input tensor.

axes [int or shape tuple, optional] Axes over which to shift. Default is None, which shifts all axes.

y [Tensor] The shifted tensor.

ifftshift : The inverse of *fftshift*.

```
>>> import mars.tensor as mt
```

```
>>> freqs = mt.fft.fftfreq(10, 0.1)
>>> freqs.execute()
array([ 0.,  1.,  2.,  3.,  4., -5., -4., -3., -2., -1.])
>>> mt.fft.fftshift(freqs).execute()
array([-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.])
```

Shift the zero-frequency component only along the second axis:

```
>>> freqs = mt.fft.fftfreq(9, d=1./9).reshape(3, 3)
>>> freqs.execute()
array([[ 0.,  1.,  2.],
       [ 3.,  4., -4.],
       [-3., -2., -1.]])
>>> mt.fft.fftshift(freqs, axes=(1,)).execute()
array([[ 2.,  0.,  1.],
       [-4.,  3.,  4.],
       [-1., -3., -2.]])
```

mars.tensor.fft.ifftshift

mars.tensor.fft.**ifftshift**(*x*, *axes=None*)

The inverse of *fftshift*. Although identical for even-length *x*, the functions differ by one sample for odd-length *x*.

x [array_like] Input tensor.

axes [int or shape tuple, optional] Axes over which to calculate. Defaults to None, which shifts all axes.

y [Tensor] The shifted tensor.

fftshift : Shift zero-frequency component to the center of the spectrum.

```
>>> import mars.tensor as mt
```

```
>>> freqs = mt.fft.fftfreq(9, d=1./9).reshape(3, 3)
>>> freqs.execute()
array([[ 0.,  1.,  2.],
       [ 3.,  4., -4.],
       [-3., -2., -1.]])
>>> mt.fft.ifftshift(mt.fft.fftshift(freqs)).execute()
array([[ 0.,  1.,  2.],
       [ 3.,  4., -4.],
       [-3., -2., -1.]])
```

2.6.5 Indexing Routines

Generating index arrays

mars.tensor.nonzero	Return the indices of the elements that are non-zero.
mars.tensor.where	Return elements, either from *x* or *y*, depending on *condition*.
mars.tensor.indices	Return a tensor representing the indices of a grid.
mars.tensor.ogrid	Construct a multi-dimensional “meshgrid”.
mars.tensor.unravel_index	Converts a flat index or tensor of flat indices into a tuple of coordinate tensors.

mars.tensor.nonzero

mars.tensor.**nonzero**(*a*)

Return the indices of the elements that are non-zero.

Returns a tuple of tensors, one for each dimension of *a*, containing the indices of the non-zero elements in that dimension. The values in *a* are always tested and returned. The corresponding non-zero values can be obtained with:

```
a[nonzero(a)]
```

To group the indices by element, rather than dimension, use:

```
transpose(nonzero(a))
```

The result of this is always a 2-D array, with a row for each non-zero element.

a [array_like] Input tensor.

tuple_of_arrays [tuple] Indices of elements that are non-zero.

flatnonzero : Return indices that are non-zero in the flattened version of the input tensor.

Tensor.nonzero : Equivalent tensor method.

count_nonzero : Counts the number of non-zero elements in the input tensor.

```
>>> import mars.tensor as mt
>>> from mars.session import new_session
```

```
>>> sess = new_session().as_default()
```

```
>>> x = mt.array([[1,0,0], [0,2,0], [1,1,0]])
>>> x.execute()
array([[1, 0, 0],
       [0, 2, 0],
       [1, 1, 0]])
>>> sess.run(mt.nonzero(x))
(array([0, 1, 2, 2]), array([0, 1, 0, 1]))
```

```
>>> x[mt.nonzero(x)].execute()  # TODO(jisheng): accomplish this after fancy␣
↪indexing is supported
```

```
>>> mt.transpose(mt.nonzero(x)).execute() # TODO(jisheng): accomplish this later
```

A common use for `nonzero` is to find the indices of an array, where a condition is True. Given an array *a*, the condition *a* > 3 is a boolean array and since False is interpreted as 0, np.nonzero(a > 3) yields the indices of the *a* where the condition is true.

```
>>> a = mt.array([[1,2,3],[4,5,6],[7,8,9]])
>>> (a > 3).execute()
array([[False, False, False],
       [ True,  True,  True],
       [ True,  True,  True]])
>>> sess.run(mt.nonzero(a > 3))
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```

The `nonzero` method of the boolean array can also be called.

```
>>> sess.run((a > 3).nonzero())
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```

mars.tensor.where

mars.tensor.**where**(*condition*, *x=None*, *y=None*)

Return elements, either from *x* or *y*, depending on *condition*.

If only *condition* is given, return `condition.nonzero()`.

condition [array_like, bool] When True, yield *x*, otherwise yield *y*.

x, y [array_like, optional] Values from which to choose. *x*, *y* and *condition* need to be broadcastable to some shape.

out [Tensor or tuple of Tensors] If both *x* and *y* are specified, the output tensor contains elements of *x* where *condition* is True, and elements from *y* elsewhere.

If only *condition* is given, return the tuple `condition.nonzero()`, the indices where *condition* is True.

nonzero, choose

If *x* and *y* are given and input arrays are 1-D, *where* is equivalent to:

```
[xv if c else yv for (c,xv,yv) in zip(condition,x,y)]
```

```
>>> import mars.tensor as mt
>>> from mars.session import new_session
```

```
>>> sess = new_session().as_default()
```

```
>>> mt.where([[True, False], [True, True]],
...          [[1, 2], [3, 4]],
...          [[9, 8], [7, 6]]).execute()
array([[1, 8],
       [3, 4]])
```

```
>>> sess.run(mt.where([[0, 1], [1, 0]]))
(array([0, 1]), array([1, 0]))
```

```
>>> x = mt.arange(9.).reshape(3, 3)
>>> sess.run(mt.where( x > 5 ))
(array([2, 2, 2]), array([0, 1, 2]))
>>> mt.where(x < 5, x, -1).execute()              # Note: broadcasting.
array([[ 0.,  1.,  2.],
       [ 3.,  4., -1.],
       [-1., -1., -1.]])
```

Find the indices of elements of *x* that are in *goodvalues*.

```
>>> goodvalues = [3, 4, 7]
>>> ix = mt.isin(x, goodvalues)
>>> ix.execute()
array([[False, False, False],
       [ True,  True, False],
       [False,  True, False]])
>>> sess.run(mt.where(ix))
(array([1, 1, 2]), array([0, 1, 1]))
```

mars.tensor.indices

mars.tensor.**indices**(*dimensions*, *dtype=<class 'int'>*, *chunk_size=None*)

Return a tensor representing the indices of a grid.

Compute a tensor where the subtensors contain index values 0,1,... varying only along the corresponding axis.

dimensions [sequence of ints] The shape of the grid.

dtype [dtype, optional] Data type of the result.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

grid [Tensor] The tensor of grid indices, `grid.shape = (len(dimensions),) + tuple(dimensions)`.

mgrid, meshgrid

The output shape is obtained by prepending the number of dimensions in front of the tuple of dimensions, i.e. if *dimensions* is a tuple `(r0, ..., rN-1)` of length `N`, the output shape is `(N,r0,...,rN-1)`.

The subtensors `grid[k]` contains the N-D array of indices along the `k-th` axis. Explicitly:

```
grid[k,i0,i1,...,iN-1] = ik
```

```
>>> import mars.tensor as mt
```

```
>>> grid = mt.indices((2, 3))
>>> grid.shape
(2, 2, 3)
>>> grid[0].execute()        # row indices
array([[0, 0, 0],
       [1, 1, 1]])
>>> grid[1].execute()        # column indices
array([[0, 1, 2],
       [0, 1, 2]])
```

The indices can be used as an index into a tensor.

```
>>> x = mt.arange(20).reshape(5, 4)
>>> row, col = mt.indices((2, 3))
>>> # x[row, col]  # TODO(jisheng): accomplish this if multiple fancy indexing is␣
↪supported
```

Note that it would be more straightforward in the above example to extract the required elements directly with `x[:2, :3]`.

mars.tensor.unravel_index

mars.tensor.**unravel_index**(*indices*, *dims*, *order='C'*)

Converts a flat index or tensor of flat indices into a tuple of coordinate tensors.

indices [array_like] An integer tensor whose elements are indices into the flattened version of a tensor of dimensions `dims`.

dims [tuple of ints] The shape of the tensor to use for unraveling `indices`.

order [{'C', 'F'}, optional] Determines whether the indices should be viewed as indexing in row-major (C-style) or column-major (Fortran-style) order.

unraveled_coords [tuple of Tensor] Each tensor in the tuple has the same shape as the `indices` tensor.

ravel_multi_index

```
>>> import mars.tensor as mt
>>> from mars.session import new_session
```

```
>>> sess = new_session().as_default()
```

```
>>> sess.run(mt.unravel_index([22, 41, 37], (7,6)))
(array([3, 6, 6]), array([4, 5, 1]))
```

```
>>> sess.run(mt.unravel_index(1621, (6,7,8,9)))
(3, 1, 4, 1)
```

Indexing-like operations

mars.tensor.take	Take elements from a tensor along an axis.
mars.tensor.choose	Construct a tensor from an index tensor and a set of tensors to choose from.
mars.tensor.compress	Return selected slices of a tensor along given axis.
mars.tensor.diag	Extract a diagonal or construct a diagonal tensor.

mars.tensor.take

mars.tensor.**take**(*a*, *indices*, *axis=None*)

Take elements from a tensor along an axis.

When axis is not None, this function does the same thing as "fancy" indexing (indexing arrays using tensors); however, it can be easier to use if you need elements along a given axis. A call such as `mt.take(arr, indices, axis=3)` is equivalent to `arr[:,:,:,indices,...]`.

Explained without fancy indexing, this is equivalent to the following use of *ndindex*, which sets each of `ii`, `jj`, and `kk` to a tuple of indices:

```
Ni, Nk = a.shape[:axis], a.shape[axis+1:]
Nj = indices.shape
for ii in ndindex(Ni):
    for jj in ndindex(Nj):
        for kk in ndindex(Nk):
            out[ii + jj + kk] = a[ii + (indices[jj],) + kk]
```

a [array_like (Ni..., M, Nk...)] The source tensor.

indices [array_like (Nj...)] The indices of the values to extract.

Also allow scalars for indices.

axis [int, optional] The axis over which to select values. By default, the flattened input tensor is used.

out [Tensor, optional (Ni..., Nj..., Nk...)] If provided, the result will be placed in this tensor. It should be of the appropriate shape and dtype.

mode [{'raise', 'wrap', 'clip'}, optional] Specifies how out-of-bounds indices will behave.

- 'raise' – raise an error (default)
- 'wrap' – wrap around
- 'clip' – clip to the range

'clip' mode means that all indices that are too large are replaced by the index that addresses the last element along that axis. Note that this disables indexing with negative numbers.

out [Tensor (Ni..., Nj..., Nk...)] The returned tensor has the same type as *a*.

compress : Take elements using a boolean mask Tensor.take : equivalent method

By eliminating the inner loop in the description above, and using *s_* to build simple slice objects, *take* can be expressed in terms of applying fancy indexing to each 1-d slice:

```
Ni, Nk = a.shape[:axis], a.shape[axis+1:]
for ii in ndindex(Ni):
    for kk in ndindex(Nj):
        out[ii + s_[...,] + kk] = a[ii + s_[:,] + kk][indices]
```

For this reason, it is equivalent to (but faster than) the following use of *apply_along_axis*:

```
out = mt.apply_along_axis(lambda a_1d: a_1d[indices], axis, a)
```

```
>>> import mars.tensor as mt
>>> a = [4, 3, 5, 7, 6, 8]
>>> indices = [0, 1, 4]
>>> mt.take(a, indices).execute()
array([4, 3, 6])
```

In this example if *a* is a tensor, "fancy" indexing can be used.

```
>>> a = mt.array(a)
>>> a[indices].execute()
array([4, 3, 6])
```

If *indices* is not one dimensional, the output also has these dimensions.

```
>>> mt.take(a, [[0, 1], [2, 3]]).execute()  # TODO(jisheng): accomplish this if␣
↪the fancy indexing is supported
```

mars.tensor.choose

mars.tensor.**choose**(*a*, *choices*, *out=None*, *mode='raise'*)

Construct a tensor from an index tensor and a set of tensors to choose from.

First of all, if confused or uncertain, definitely look at the Examples - in its full generality, this function is less simple than it might seem from the following code description (below ndi = *mt.lib.index_tricks*):

`mt.choose(a,c) == mt.array([c[a[I]][I] for I in ndi.ndindex(a.shape)]).`

But this omits some subtleties. Here is a fully general summary:

Given an "index" tensor (*a*) of integers and a sequence of *n* tensors (*choices*), *a* and each choice tensor are first broadcast, as necessary, to tensors of a common shape; calling these *Ba* and *Bchoices[i], i = 0,...,n-1* we have that, necessarily, `Ba.shape == Bchoices[i].shape` for each *i*. Then, a new array with shape `Ba.shape` is created as follows:

- if `mode=raise` (the default), then, first of all, each element of *a* (and thus *Ba*) must be in the range *[0, n-1]*; now, suppose that *i* (in that range) is the value at the *(j0, j1, ..., jm)* position in *Ba* - then the value at the same position in the new array is the value in *Bchoices[i]* at that same position;
- if `mode=wrap`, values in *a* (and thus *Ba*) may be any (signed) integer; modular arithmetic is used to map integers outside the range *[0, n-1]* back into that range; and then the new array is constructed as above;
- if `mode=clip`, values in *a* (and thus *Ba*) may be any (signed) integer; negative integers are mapped to 0; values greater than *n-1* are mapped to *n-1*; and then the new tensor is constructed as above.

a [int tensor] This tensor must contain integers in *[0, n-1]*, where *n* is the number of choices, unless `mode=wrap` or `mode=clip`, in which cases any integers are permissible.

choices [sequence of tensors] Choice tensors. *a* and all of the choices must be broadcastable to the same shape. If *choices* is itself a tensor (not recommended), then its outermost dimension (i.e., the one corresponding to `choices.shape[0]`) is taken as defining the "sequence".

out [tensor, optional] If provided, the result will be inserted into this tensor. It should be of the appropriate shape and dtype.

mode [{'raise' (default), 'wrap', 'clip'}, optional] Specifies how indices outside *[0, n-1]* will be treated:

- 'raise' : an exception is raised
- 'wrap' : value becomes value mod *n*
- 'clip' : values < 0 are mapped to 0, values > n-1 are mapped to n-1

merged_array [Tensor] The merged result.

ValueError: shape mismatch If *a* and each choice tensor are not all broadcastable to the same shape.

Tensor.choose : equivalent method

To reduce the chance of misinterpretation, even though the following "abuse" is nominally supported, *choices* should neither be, nor be thought of as, a single tensor, i.e., the outermost sequence-like container should be either a list or a tuple.

```
>>> import mars.tensor as mt
```

```
>>> choices = [[0, 1, 2, 3], [10, 11, 12, 13],
...   [20, 21, 22, 23], [30, 31, 32, 33]]
>>> mt.choose([2, 3, 1, 0], choices
... # the first element of the result will be the first element of the
... # third (2+1) "array" in choices, namely, 20; the second element
... # will be the second element of the fourth (3+1) choice array, i.e.,
... # 31, etc.
... ).execute()
array([20, 31, 12,  3])
>>> mt.choose([2, 4, 1, 0], choices, mode='clip').execute() # 4 goes to 3 (4-1)
array([20, 31, 12,  3])
>>> # because there are 4 choice arrays
>>> mt.choose([2, 4, 1, 0], choices, mode='wrap').execute() # 4 goes to (4 mod 4)
array([20,  1, 12,  3])
>>> # i.e., 0
```

A couple examples illustrating how choose broadcasts:

```
>>> a = [[1, 0, 1], [0, 1, 0], [1, 0, 1]]
>>> choices = [-10, 10]
>>> mt.choose(a, choices).execute()
array([[ 10, -10,  10],
       [-10,  10, -10],
       [ 10, -10,  10]])
```

```
>>> # With thanks to Anne Archibald
>>> a = mt.array([0, 1]).reshape((2,1,1))
>>> c1 = mt.array([1, 2, 3]).reshape((1,3,1))
>>> c2 = mt.array([-1, -2, -3, -4, -5]).reshape((1,1,5))
>>> mt.choose(a, (c1, c2)).execute() # result is 2x3x5, res[0,:,:]=c1, res[1,:,
↪:]=c2
array([[[ 1,  1,  1,  1,  1],
        [ 2,  2,  2,  2,  2],
        [ 3,  3,  3,  3,  3]],
       [[-1, -2, -3, -4, -5],
        [-1, -2, -3, -4, -5],
        [-1, -2, -3, -4, -5]]])
```

mars.tensor.compress

mars.tensor.**compress**(*condition*, *a*, *axis=None*, *out=None*)

Return selected slices of a tensor along given axis.

When working along a given axis, a slice along that axis is returned in *output* for each index where *condition* evaluates to True. When working on a 1-D array, *compress* is equivalent to *extract*.

condition [1-D tensor of bools] Tensor that selects which entries to return. If len(condition) is less than the size of *a* along the given axis, then output is truncated to the length of the condition tensor.

a [array_like] Tensor from which to extract a part.

axis [int, optional] Axis along which to take slices. If None (default), work on the flattened tensor.

out [Tensor, optional] Output tensor. Its type is preserved and it must be of the right shape to hold the output.

compressed_array [Tensor] A copy of *a* without the slices along axis for which *condition* is false.

take, choose, diag, diagonal, select Tensor.compress : Equivalent method in ndarray mt.extract: Equivalent method when working on 1-D arrays

```
>>> import mars.tensor as mt
```

```
>>> a = mt.array([[1, 2], [3, 4], [5, 6]])
>>> a.execute()
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> mt.compress([0, 1], a, axis=0).execute()
array([[3, 4]])
>>> mt.compress([False, True, True], a, axis=0).execute()
array([[3, 4],
       [5, 6]])
```

(continues on next page)

(continued from previous page)

```
>>> mt.compress([False, True], a, axis=1).execute()
array([[2],
       [4],
       [6]])
```

Working on the flattened tensor does not return slices along an axis but selects elements.

```
>>> mt.compress([False, True], a).execute()
array([2])
```

2.6.6 Linear Algebra

Matrix and vector products

mars.tensor.dot	Dot product of two arrays.
mars.tensor.vdot	Return the dot product of two vectors.
mars.tensor.inner	Returns the inner product of a and b for arrays of floating point types.
mars.tensor.matmul	Matrix product of two tensors.
mars.tensor.tensordot	Compute tensor dot product along specified axes for tensors >= 1-D.

mars.tensor.dot

mars.tensor.**dot**(*a*, *b*, *out=None*, *sparse=None*)

Dot product of two arrays. Specifically,

- If both *a* and *b* are 1-D arrays, it is inner product of vectors (without complex conjugation).
- If both *a* and *b* are 2-D arrays, it is matrix multiplication, but using *matmul()* or `a @ b` is preferred.
- If either *a* or *b* is 0-D (scalar), it is equivalent to *multiply()* and using `numpy.multiply(a, b)` or `a * b` is preferred.
- If *a* is an N-D array and *b* is a 1-D array, it is a sum product over the last axis of *a* and *b*.
- If *a* is an N-D array and *b* is an M-D array (where `M>=2`), it is a sum product over the last axis of *a* and the second-to-last axis of *b*:

```
dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])
```

a [array_like] First argument.

b [array_like] Second argument.

out [Tensor, optional] Output argument. This must have the exact kind that would be returned if it was not used. In particular, it must have the right type, must be C-contiguous, and its dtype must be the dtype that would be returned for *dot(a,b)*. This is a performance feature. Therefore, if these conditions are not met, an exception is raised, instead of attempting to be flexible.

output [Tensor] Returns the dot product of *a* and *b*. If *a* and *b* are both scalars or both 1-D arrays then a scalar is returned; otherwise a tensor is returned. If *out* is given, then it is returned.

ValueError If the last dimension of *a* is not the same size as the second-to-last dimension of *b*.

vdot : Complex-conjugating dot product. tensordot : Sum products over arbitrary axes. einsum : Einstein summation convention. matmul : '@' operator as method with out parameter.

```
>>> import mars.tensor as mt
```

```
>>> mt.dot(3, 4).execute()
12
```

Neither argument is complex-conjugated:

```
>>> mt.dot([2j, 3j], [2j, 3j]).execute()
(-13+0j)
```

For 2-D arrays it is the matrix product:

```
>>> a = [[1, 0], [0, 1]]
>>> b = [[4, 1], [2, 2]]
>>> mt.dot(a, b).execute()
array([[4, 1],
       [2, 2]])
```

```
>>> a = mt.arange(3*4*5*6).reshape((3,4,5,6))
>>> b = mt.arange(3*4*5*6)[::-1].reshape((5,4,6,3))
>>> mt.dot(a, b)[2,3,2,1,2,2].execute()
499128
>>> mt.sum(a[2,3,2,:] * b[1,2,:,2]).execute()
499128
```

mars.tensor.vdot

mars.tensor.**vdot**(*a*, *b*)

Return the dot product of two vectors.

The vdot(*a*, *b*) function handles complex numbers differently than dot(*a*, *b*). If the first argument is complex the complex conjugate of the first argument is used for the calculation of the dot product.

Note that *vdot* handles multidimensional tensors differently than *dot*: it does *not* perform a matrix product, but flattens input arguments to 1-D vectors first. Consequently, it should only be used for vectors.

a [array_like] If *a* is complex the complex conjugate is taken before calculation of the dot product.

b [array_like] Second argument to the dot product.

output [Tensor] Dot product of *a* and *b*. Can be an int, float, or complex depending on the types of *a* and *b*.

dot [Return the dot product without using the complex conjugate of the] first argument.

```
>>> import mars.tensor as mt
```

```
>>> a = mt.array([1+2j,3+4j])
>>> b = mt.array([5+6j,7+8j])
>>> mt.vdot(a, b).execute()
(70-8j)
>>> mt.vdot(b, a).execute()
(70+8j)
```

Note that higher-dimensional arrays are flattened!

```
>>> a = mt.array([[1, 4], [5, 6]])
>>> b = mt.array([[4, 1], [2, 2]])
>>> mt.vdot(a, b).execute()
30
>>> mt.vdot(b, a).execute()
30
>>> 1*4 + 4*1 + 5*2 + 6*2
30
```

mars.tensor.inner

mars.tensor.**inner**(*a*, *b*, *sparse=None*)

Returns the inner product of a and b for arrays of floating point types.

Like the generic NumPy equivalent the product sum is over the last dimension of a and b. The first argument is not conjugated.

mars.tensor.matmul

mars.tensor.**matmul**(*a*, *b*, *sparse=None*, *out=None*, ***kw*)

Matrix product of two tensors.

The behavior depends on the arguments in the following way.

- If both arguments are 2-D they are multiplied like conventional matrices.
- If either argument is N-D, N > 2, it is treated as a stack of matrices residing in the last two indexes and broadcast accordingly.
- If the first argument is 1-D, it is promoted to a matrix by prepending a 1 to its dimensions. After matrix multiplication the prepended 1 is removed.
- If the second argument is 1-D, it is promoted to a matrix by appending a 1 to its dimensions. After matrix multiplication the appended 1 is removed.

Multiplication by a scalar is not allowed, use $*$ instead. Note that multiplying a stack of matrices with a vector will result in a stack of vectors, but matmul will not recognize it as such.

`matmul` differs from `dot` in two important ways.

- Multiplication by scalars is not allowed.
- Stacks of matrices are broadcast together as if the matrices were elements.

a [array_like] First argument.

b [array_like] Second argument.

out [Tensor, optional] Output argument. This must have the exact kind that would be returned if it was not used. In particular, it must have the right type, and its dtype must be the dtype that would be returned for *dot(a,b)*. This is a performance feature. Therefore, if these conditions are not met, an exception is raised, instead of attempting to be flexible.

output [Tensor] Returns the dot product of *a* and *b*. If *a* and *b* are both 1-D arrays then a scalar is returned; otherwise an array is returned. If *out* is given, then it is returned.

ValueError If the last dimension of *a* is not the same size as the second-to-last dimension of *b*.

If scalar value is passed.

vdot : Complex-conjugating dot product. tensordot : Sum products over arbitrary axes. dot : alternative matrix product with different broadcasting rules.

The matmul function implements the semantics of the @ operator introduced in Python 3.5 following PEP465.

For 2-D arrays it is the matrix product:

```
>>> import mars.tensor as mt
```

```
>>> a = [[1, 0], [0, 1]]
>>> b = [[4, 1], [2, 2]]
>>> mt.matmul(a, b).execute()
array([[4, 1],
       [2, 2]])
```

For 2-D mixed with 1-D, the result is the usual.

```
>>> a = [[1, 0], [0, 1]]
>>> b = [1, 2]
>>> mt.matmul(a, b).execute()
array([1, 2])
>>> mt.matmul(b, a).execute()
array([1, 2])
```

Broadcasting is conventional for stacks of arrays

```
>>> a = mt.arange(2*2*4).reshape((2,2,4))
>>> b = mt.arange(2*2*4).reshape((2,4,2))
>>> mt.matmul(a,b).shape
(2, 2, 2)
>>> mt.matmul(a,b)[0,1,1].execute()
98
>>> mt.sum(a[0,1,:] * b[0,:,1]).execute()
98
```

Vector, vector returns the scalar inner product, but neither argument is complex-conjugated:

```
>>> mt.matmul([2j, 3j], [2j, 3j]).execute()
(-13+0j)
```

Scalar multiplication raises an error.

```
>>> mt.matmul([1,2], 3)
Traceback (most recent call last):
...
ValueError: Scalar operands are not allowed, use '*' instead
```

mars.tensor.tensordot

`mars.tensor.`**`tensordot`**(*a*, *b*, *axes=2*, *sparse=None*)

Compute tensor dot product along specified axes for tensors >= 1-D. Given two tensors (arrays of dimension greater than or equal to one), *a* and *b*, and an array_like object containing two array_like objects, `(a_axes, b_axes)`, sum the products of *a*'s and *b*'s elements (components) over the axes specified by `a_axes` and

`b_axes`. The third argument can be a single non-negative integer_like scalar, `N`; if it is such, then the last `N` dimensions of *a* and the first `N` dimensions of *b* are summed over. Parameters ———- a, b : array_like, len(shape) >= 1

Tensors to "dot".

axes [int or (2,) array_like]

- integer_like If an int N, sum over the last N axes of *a* and the first N axes of *b* in order. The sizes of the corresponding axes must match.
- (2,) array_like Or, a list of axes to be summed over, first sequence applying to *a*, second to *b*. Both elements array_like must be of the same length.

dot, einsum Notes —– Three common use cases are:

- `axes = 0` : tensor product $a \otimes b$
- `axes = 1` : tensor dot product $a \cdot b$
- `axes = 2` : (default) tensor double contraction $a : b$

When *axes* is integer_like, the sequence for evaluation will be: first the -Nth axis in *a* and 0th axis in *b*, and the -1th axis in *a* and Nth axis in *b* last. When there is more than one axis to sum over - and they are not the last (first) axes of *a* (*b*) - the argument *axes* should consist of two sequences of the same length, with the first axis to sum over given first in both sequences, the second axis second, and so forth. Examples ——— >>> import mars.tensor as mt

A "traditional" example: >>> a = mt.arange(60.).reshape(3,4,5) >>> b = mt.arange(24.).reshape(4,3,2) >>> c = mt.tensordot(a,b, axes=([1,0],[0,1])) >>> c.shape (5, 2) >>> r = c.execute() >>> r array([[4400., 4730.],

[4532., 4874.], [4664., 5018.], [4796., 5162.], [4928., 5306.]])

```
>>> # A slower but equivalent way of computing the same...
>>> ra = np.arange(60.).reshape(3,4,5)
>>> rb = np.arange(24.).reshape(4,3,2)
>>> d = np.zeros((5,2))
>>> for i in range(5):
...   for j in range(2):
...     for k in range(3):
...       for n in range(4):
...         d[i,j] += ra[k,n,i] * rb[n,k,j]
>>> r == d
array([[ True,  True],
       [ True,  True],
       [ True,  True],
       [ True,  True],
       [ True,  True]], dtype=bool)
An extended example taking advantage of the overloading of + and \*:
>>> a = mt.array(range(1, 9))
>>> a.shape = (2, 2, 2)
>>> A = mt.array(('a', 'b', 'c', 'd'), dtype=object)
>>> A.shape = (2, 2)
>>> a.execute(); A.execute()
array([[[1, 2],
        [3, 4]],
       [[5, 6],
        [7, 8]]])
array([[a, b],
       [c, d]], dtype=object)
```

(continues on next page)

(continued from previous page)

```
>>> mt.tensordot(a, A).execute() # third argument default is 2 for double-
↪contraction
array([abbcccdddd, aaaaabbbbbbccccccddddddddd], dtype=object)
>>> mt.tensordot(a, A, 1).execute()
array([[[acc, bdd],
        [aaacccc, bbbdddd]],
       [[aaaaacccccc, bbbbbddddddd],
        [aaaaaaaccccccccc, bbbbbbbddddddddd]]], dtype=object)
>>> mt.tensordot(a, A, 0).execute() # tensor product (result too long to incl.)
array([[[[[a, b],
          [c, d]],
          ...
>>> mt.tensordot(a, A, (0, 1)).execute()
array([[[abbbbb, cddddd],
        [aabbbbbb, ccdddddd]],
       [[aaabbbbbbb, cccddddddd],
        [aaaabbbbbbbb, ccccddddddddd]]], dtype=object)
>>> mt.tensordot(a, A, (2, 1)).execute()
array([[[abb, cdd],
        [aaabbbb, cccdddd]],
       [[aaaaabbbbbb, ccccccddddddd],
        [aaaaaaabbbbbbbb, ccccccccddddddddd]]], dtype=object)
>>> mt.tensordot(a, A, ((0, 1), (0, 1))).execute()
array([abbbccccccddddddd, aabbbbccccccddddddddd], dtype=object)
>>> mt.tensordot(a, A, ((2, 1), (1, 0))).execute()
array([acccbbdddd, aaaaaccccccbbbbbbddddddddd], dtype=object)
```

Decompositions

mars.tensor.linalg.cholesky	Cholesky decomposition.
mars.tensor.linalg.qr	Compute the qr factorization of a matrix.
mars.tensor.linalg.svd	Singular Value Decomposition.

mars.tensor.linalg.cholesky

mars.tensor.linalg.**cholesky**(*a*, *lower=False*)

Cholesky decomposition.

Return the Cholesky decomposition, *L* * *L.H*, of the square matrix *a*, where *L* is lower-triangular and .H is the conjugate transpose operator (which is the ordinary transpose if *a* is real-valued). *a* must be Hermitian (symmetric if real-valued) and positive-definite. Only *L* is actually returned.

a [(..., M, M) array_like] Hermitian (symmetric if all elements are real), positive-definite input matrix.

lower [bool] Whether to compute the upper or lower triangular Cholesky factorization. Default is upper-triangular.

L [(..., M, M) array_like] Upper or lower-triangular Cholesky factor of *a*.

LinAlgError If the decomposition fails, for example, if *a* is not positive-definite.

Broadcasting rules apply, see the *mt.linalg* documentation for details.

The Cholesky decomposition is often used as a fast way of solving

$$A\mathbf{x} = \mathbf{b}$$

(when A is both Hermitian/symmetric and positive-definite).

First, we solve for $\mathbf{y}$ in

$$L\mathbf{y} = \mathbf{b},$$

and then for $\mathbf{x}$ in

$$L.H\mathbf{x} = \mathbf{y}.$$

```
>>> import mars.tensor as mt
```

```
>>> A = mt.array([[1,-2j],[2j,5]])
>>> A.execute()
array([[ 1.+0.j,  0.-2.j],
       [ 0.+2.j,  5.+0.j]])
>>> L = mt.linalg.cholesky(A, lower=True)
>>> L.execute()
array([[ 1.+0.j,  0.+0.j],
       [ 0.+2.j,  1.+0.j]])
>>> mt.dot(L, L.T.conj()).execute() # verify that L * L.H = A
array([[ 1.+0.j,  0.-2.j],
       [ 0.+2.j,  5.+0.j]])
>>> A = [[1,-2j],[2j,5]] # what happens if A is only array_like?
>>> mt.linalg.cholesky(A, lower=True).execute()
array([[ 1.+0.j,  0.+0.j],
       [ 0.+2.j,  1.+0.j]])
```

mars.tensor.linalg.qr

mars.tensor.linalg.**qr**(*a*, *method='tsqr'*)

Compute the qr factorization of a matrix.

Factor the matrix *a* as *qr*, where *q* is orthonormal and *r* is upper-triangular.

a [array_like, shape (M, N)] Matrix to be factored.

method: {'tsqr', 'sfqr'}, optional method to calculate qr factorization, tsqr as default

TSQR is presented in:

A. Benson, D. Gleich, and J. Demmel. Direct QR factorizations for tall-and-skinny matrices in MapReduce architectures. IEEE International Conference on Big Data, 2013. http://arxiv.org/abs/1301.1071

FSQR is a QR decomposition for fat and short matrix: A = [A1, A2, A3, ...], A1 may be decomposed as A1 = Q1 * R1, for A = Q * R, Q = Q1, R = [R1, R2, R3, ...] where A2 = Q1 * R2, A3 = Q1 * R3, ...

q [Tensor of float or complex, optional] A matrix with orthonormal columns. When mode = 'complete' the result is an orthogonal/unitary matrix depending on whether or not a is real/complex. The determinant may be either +/- 1 in that case.

r [Tensor of float or complex, optional] The upper-triangular matrix.

LinAlgError If factoring fails.

For more information on the qr factorization, see for example: http://en.wikipedia.org/wiki/QR_factorization

```
>>> import mars.tensor as mt
```

```
>>> a = mt.random.randn(9, 6)
>>> q, r = mt.linalg.qr(a)
>>> mt.allclose(a, mt.dot(q, r)).execute()  # a does equal qr
True
```

mars.tensor.linalg.svd

`mars.tensor.linalg.`**`svd`**(*a*, *method='tsqr'*)

Singular Value Decomposition.

When *a* is a 2D tensor, it is factorized as `u @ np.diag(s) @ vh = (u * s) @ vh`, where *u* and *vh* are 2D unitary tensors and *s* is a 1D tensor of *a*'s singular values. When *a* is higher-dimensional, SVD is applied in stacked mode as explained below.

a [(..., M, N) array_like] A real or complex tensor with `a.ndim >= 2`.

method: {'tsqr'}, optional method to calculate qr factorization, tsqr as default

TSQR is presented in:

A. Benson, D. Gleich, and J. Demmel. Direct QR factorizations for tall-and-skinny matrices in MapReduce architectures. IEEE International Conference on Big Data, 2013. http://arxiv.org/abs/1301.1071

u [{ (..., M, M), (..., M, K) } tensor] Unitary tensor(s). The first `a.ndim - 2` dimensions have the same size as those of the input *a*. The size of the last two dimensions depends on the value of *full_matrices*. Only returned when *compute_uv* is True.

s [(..., K) tensor] Vector(s) with the singular values, within each vector sorted in descending order. The first `a.ndim - 2` dimensions have the same size as those of the input *a*.

vh [{ (..., N, N), (..., K, N) } tensor] Unitary tensor(s). The first `a.ndim - 2` dimensions have the same size as those of the input *a*. The size of the last two dimensions depends on the value of *full_matrices*. Only returned when *compute_uv* is True.

LinAlgError If SVD computation does not converge.

SVD is usually described for the factorization of a 2D matrix A. The higher-dimensional case will be discussed below. In the 2D case, SVD is written as $A = USV^H$, where $A = a$, $U = u$, $S = \mathtt{np.diag}(s)$ and $V^H = vh$. The 1D tensor *s* contains the singular values of *a* and *u* and *vh* are unitary. The rows of *vh* are the eigenvectors of $A^H A$ and the columns of *u* are the eigenvectors of AA^H. In both cases the corresponding (possibly non-zero) eigenvalues are given by `s**2`.

If *a* has more than two dimensions, then broadcasting rules apply, as explained in routines.linalg-broadcasting. This means that SVD is working in "stacked" mode: it iterates over all indices of the first `a.ndim - 2` dimensions and for each combination SVD is applied to the last two indices. The matrix *a* can be reconstructed from the decomposition with either `(u * s[..., None, :]) @ vh` or `u @ (s[..., None] * vh)`. (The `@` operator can be replaced by the function `mt.matmul` for python versions below 3.5.)

```
>>> import mars.tensor as mt
>>> a = mt.random.randn(9, 6) + 1j*mt.random.randn(9, 6)
>>> b = mt.random.randn(2, 7, 8, 3) + 1j*mt.random.randn(2, 7, 8, 3)
```

Reconstruction based on reduced SVD, 2D case:

```
>>> u, s, vh = mt.linalg.svd(a)
>>> u.shape, s.shape, vh.shape
((9, 6), (6,), (6, 6))
>>> np.allclose(a, np.dot(u * s, vh))
True
>>> smat = np.diag(s)
>>> np.allclose(a, np.dot(u, np.dot(smat, vh)))
True
```

Norms and other numbers

mars.tensor.linalg.norm	Matrix or vector norm.

mars.tensor.linalg.norm

mars.tensor.linalg.**norm**(*x*, *ord=None*, *axis=None*, *keepdims=False*)

Matrix or vector norm.

This function is able to return one of eight different matrix norms, or one of an infinite number of vector norms (described below), depending on the value of the `ord` parameter.

x [array_like] Input tensor. If *axis* is None, *x* must be 1-D or 2-D.

ord [{non-zero int, inf, -inf, 'fro', 'nuc'}, optional] Order of the norm (see table under `Notes`). inf means mars tensor's *inf* object.

axis [{int, 2-tuple of ints, None}, optional] If *axis* is an integer, it specifies the axis of *x* along which to compute the vector norms. If *axis* is a 2-tuple, it specifies the axes that hold 2-D matrices, and the matrix norms of these matrices are computed. If *axis* is None then either a vector norm (when *x* is 1-D) or a matrix norm (when *x* is 2-D) is returned.

keepdims [bool, optional] If this is set to True, the axes which are normed over are left in the result as dimensions with size one. With this option the result will broadcast correctly against the original *x*.

n [float or Tensor] Norm of the matrix or vector(s).

For values of `ord <= 0`, the result is, strictly speaking, not a mathematical 'norm', but it may still be useful for various numerical purposes.

The following norms can be calculated:

ord	norm for matrices	norm for vectors
None	Frobenius norm	2-norm
'fro'	Frobenius norm	–
'nuc'	nuclear norm	–
inf	max(sum(abs(x), axis=1))	max(abs(x))
-inf	min(sum(abs(x), axis=1))	min(abs(x))
0	–	sum(x != 0)
1	max(sum(abs(x), axis=0))	as below
-1	min(sum(abs(x), axis=0))	as below
2	2-norm (largest sing. value)	as below
-2	smallest singular value	as below
other	–	sum(abs(x)**ord)**(1./ord)

The Frobenius norm is given by[1]:

$||A||_F = [$
$sum_{i,j} abs(a_{i,j})^2]^{1/2}$

The nuclear norm is the sum of the singular values.

```
>>> from mars.tensor import linalg as LA
>>> import mars.tensor as mt
>>> a = mt.arange(9) - 4
>>> a.execute()
array([-4, -3, -2, -1,  0,  1,  2,  3,  4])
>>> b = a.reshape((3, 3))
>>> b.execute()
array([[-4, -3, -2],
       [-1,  0,  1],
       [ 2,  3,  4]])
```

```
>>> LA.norm(a).execute()
7.745966692414834
>>> LA.norm(b).execute()
7.745966692414834
>>> LA.norm(b, 'fro').execute()
7.745966692414834
>>> LA.norm(a, mt.inf).execute()
4.0
>>> LA.norm(b, mt.inf).execute()
9.0
>>> LA.norm(a, -mt.inf).execute()
0.0
>>> LA.norm(b, -mt.inf).execute()
2.0
```

```
>>> LA.norm(a, 1).execute()
20.0
>>> LA.norm(b, 1).execute()
7.0
>>> LA.norm(a, -1).execute()
0.0
>>> LA.norm(b, -1).execute()
6.0
```

(continues on next page)

[1] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Baltimore, MD, Johns Hopkins University Press, 1985, pg. 15

(continued from previous page)

```
>>> LA.norm(a, 2).execute()
7.745966692414834
>>> LA.norm(b, 2).execute()
7.3484692283495345
```

```
>>> LA.norm(a, -2).execute()
0.0
>>> LA.norm(b, -2).execute()
4.351066026358965e-18
>>> LA.norm(a, 3).execute()
5.8480354764257312
>>> LA.norm(a, -3).execute()
0.0
```

Using the *axis* argument to compute vector norms:

```
>>> c = mt.array([[ 1, 2, 3],
...               [-1, 1, 4]])
>>> LA.norm(c, axis=0).execute()
array([ 1.41421356,  2.23606798,  5.        ])
>>> LA.norm(c, axis=1).execute()
array([ 3.74165739,  4.24264069])
>>> LA.norm(c, ord=1, axis=1).execute()
array([ 6.,  6.])
```

Using the *axis* argument to compute matrix norms:

```
>>> m = mt.arange(8).reshape(2,2,2)
>>> LA.norm(m, axis=(1,2)).execute()
array([  3.74165739,  11.22497216])
>>> LA.norm(m[0, :, :]).execute(), LA.norm(m[1, :, :]).execute()
(3.7416573867739413, 11.224972160321824)
```

2.6.7 Logic Functions

Truth value testing

mars.tensor.all	Test whether all array elements along a given axis evaluate to True.
mars.tensor.any	Test whether any tensor element along a given axis evaluates to True.

mars.tensor.all

mars.tensor.**all**(*a*, *axis=None*, *out=None*, *keepdims=None*, *combine_size=None*)

Test whether all array elements along a given axis evaluate to True.

a [array_like] Input tensor or object that can be converted to a tensor.

axis [None or int or tuple of ints, optional] Axis or axes along which a logical AND reduction is performed. The default (*axis = None*) is to perform a logical AND over all the dimensions of the input array. *axis* may be negative, in which case it counts from the last to the first axis.

If this is a tuple of ints, a reduction is performed on multiple axes, instead of a single axis or all the axes as before.

out [Tensor, optional] Alternate output tensor in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if `dtype(out)` is float, the result will consist of 0.0's and 1.0's). See *doc.ufuncs* (Section "Output arguments") for more details.

keepdims [bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input tensor.

If the default value is passed, then *keepdims* will not be passed through to the *all* method of sub-classes of *ndarray*, however any non-default value will be. If the sub-classes *sum* method does not implement *keepdims* any exceptions will be raised.

combine_size: int, optional The number of chunks to combine.

all [Tensor, bool] A new boolean or tensor is returned unless *out* is specified, in which case a reference to *out* is returned.

Tensor.all : equivalent method

any : Test whether any element along a given axis evaluates to True.

Not a Number (NaN), positive infinity and negative infinity evaluate to *True* because these are not equal to zero.

```
>>> import mars.tensor as mt
```

```
>>> mt.all([[True,False],[True,True]]).execute()
False
```

```
>>> mt.all([[True,False],[True,True]], axis=0).execute()
array([ True, False])
```

```
>>> mt.all([-1, 4, 5]).execute()
True
```

```
>>> mt.all([1.0, mt.nan]).execute()
True
```

mars.tensor.any

mars.tensor.**any**(*a*, *axis=None*, *out=None*, *keepdims=None*, *combine_size=None*)

Test whether any tensor element along a given axis evaluates to True.

Returns single boolean unless *axis* is not `None`

a [array_like] Input tensor or object that can be converted to an array.

axis [None or int or tuple of ints, optional] Axis or axes along which a logical OR reduction is performed. The default (*axis* = *None*) is to perform a logical OR over all the dimensions of the input array. *axis* may be negative, in which case it counts from the last to the first axis.

If this is a tuple of ints, a reduction is performed on multiple axes, instead of a single axis or all the axes as before.

out [Tensor, optional] Alternate output tensor in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if it is of type float, then it will remain so, returning 1.0

for True and 0.0 for False, regardless of the type of *a*). See *doc.ufuncs* (Section "Output arguments") for details.

keepdims [bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input tensor.

If the default value is passed, then *keepdims* will not be passed through to the *any* method of sub-classes of *Tensor*, however any non-default value will be. If the sub-classes *sum* method does not implement *keepdims* any exceptions will be raised.

combine_size: int, optional The number of chunks to combine.

any [bool or Tensor] A new boolean or *Tensor* is returned unless *out* is specified, in which case a reference to *out* is returned.

Tensor.any : equivalent method

all : Test whether all elements along a given axis evaluate to True.

Not a Number (NaN), positive infinity and negative infinity evaluate to *True* because these are not equal to zero.

```
>>> import mars.tensor as mt
```

```
>>> mt.any([[True, False], [True, True]]).execute()
True
```

```
>>> mt.any([[True, False], [False, False]], axis=0).execute()
array([ True, False])
```

```
>>> mt.any([-1, 0, 5]).execute()
True
```

```
>>> mt.any(mt.nan).execute()
True
```

Array contents

mars.tensor.isfinite	Test element-wise for finiteness (not infinity or not Not a Number).
mars.tensor.isinf	Test element-wise for positive or negative infinity.
mars.tensor.isnan	Test element-wise for NaN and return result as a boolean tensor.

Array type testing

mars.tensor.iscomplex	Returns a bool tensor, where True if input element is complex.
mars.tensor.isreal	Returns a bool tensor, where True if input element is real.

mars.tensor.iscomplex

mars.tensor.**iscomplex**(*x*, ***kwargs*)

Returns a bool tensor, where True if input element is complex.

What is tested is whether the input has a non-zero imaginary part, not if the input type is complex.

x [array_like] Input tensor.

out [Tensor of bools] Output tensor.

isreal iscomplexobj : Return True if x is a complex type or an array of complex

numbers.

```
>>> import mars.tensor as mt
```

```
>>> mt.iscomplex([1+1j, 1+0j, 4.5, 3, 2, 2j]).execute()
array([ True, False, False, False, False,  True])
```

mars.tensor.isreal

mars.tensor.**isreal**(*x*, ***kwargs*)

Returns a bool tensor, where True if input element is real.

If element has complex type with zero complex part, the return value for that element is True.

x [array_like] Input tensor.

out [Tensor, bool] Boolean tensor of same shape as *x*.

iscomplex isrealobj : Return True if x is not a complex type.

```
>>> import mars.tensor as mt
```

```
>>> mt.isreal([1+1j, 1+0j, 4.5, 3, 2, 2j]).execute()
array([False,  True,  True,  True,  True, False])
```

Logic operations

mars.tensor.logical_and	Compute the truth value of x1 AND x2 element-wise.
mars.tensor.logical_or	Compute the truth value of x1 OR x2 element-wise.
mars.tensor.logical_not	Compute the truth value of NOT x element-wise.
mars.tensor.logical_xor	Compute the truth value of x1 XOR x2, element-wise.

Comparison

mars.tensor.allclose	Returns True if two tensors are element-wise equal within a tolerance.
mars.tensor.isclose	Returns a boolean tensor where two tensors are element-wise equal within a tolerance.

Continued on next page

Table 34 – continued from previous page

mars.tensor.array_equal	True if two tensors have the same shape and elements, False otherwise.
mars.tensor.greater	Return the truth value of (x1 > x2) element-wise.
mars.tensor.greater_equal	Return the truth value of (x1 >= x2) element-wise.
mars.tensor.less	Return the truth value of (x1 < x2) element-wise.
mars.tensor.less_equal	Return the truth value of (x1 =< x2) element-wise.
mars.tensor.equal	Return (x1 == x2) element-wise.
mars.tensor.not_equal	Return (x1 != x2) element-wise.

mars.tensor.allclose

mars.tensor.**allclose**(*a*, *b*, *rtol=1e-05*, *atol=1e-08*, *equal_nan=False*)

Returns True if two tensors are element-wise equal within a tolerance.

The tolerance values are positive, typically very small numbers. The relative difference (*rtol* * abs(*b*)) and the absolute difference *atol* are added together to compare against the absolute difference between *a* and *b*.

If either array contains one or more NaNs, False is returned. Infs are treated as equal if they are in the same place and of the same sign in both tensors.

a, b [array_like] Input tensors to compare.

rtol [float] The relative tolerance parameter (see Notes).

atol [float] The absolute tolerance parameter (see Notes).

equal_nan [bool] Whether to compare NaN's as equal. If True, NaN's in *a* will be considered equal to NaN's in *b* in the output tensor.

allclose [bool] Returns True if the two tensors are equal within the given tolerance; False otherwise.

isclose, all, any, equal

If the following equation is element-wise True, then allclose returns True.

absolute(*a* - *b*) <= (*atol* + *rtol* * absolute(*b*))

The above equation is not symmetric in *a* and *b*, so that `allclose(a, b)` might be different from `allclose(b, a)` in some rare cases.

The comparison of *a* and *b* uses standard broadcasting, which means that *a* and *b* need not have the same shape in order for `allclose(a, b)` to evaluate to True. The same is true for *equal* but not *array_equal*.

```
>>> import mars.tensor as mt
```

```
>>> mt.allclose([1e10,1e-7], [1.00001e10,1e-8]).execute()
False
>>> mt.allclose([1e10,1e-8], [1.00001e10,1e-9]).execute()
True
>>> mt.allclose([1e10,1e-8], [1.0001e10,1e-9]).execute()
False
>>> mt.allclose([1.0, mt.nan], [1.0, mt.nan]).execute()
False
>>> mt.allclose([1.0, mt.nan], [1.0, mt.nan], equal_nan=True).execute()
True
```

mars.tensor.isclose

mars.tensor.**isclose**(*a*, *b*, *rtol=1e-05*, *atol=1e-08*, *equal_nan=False*)

Returns a boolean tensor where two tensors are element-wise equal within a tolerance.

The tolerance values are positive, typically very small numbers. The relative difference (*rtol* * abs(*b*)) and the absolute difference *atol* are added together to compare against the absolute difference between *a* and *b*.

a, b [array_like] Input tensors to compare.

rtol [float] The relative tolerance parameter (see Notes).

atol [float] The absolute tolerance parameter (see Notes).

equal_nan [bool] Whether to compare NaN's as equal. If True, NaN's in *a* will be considered equal to NaN's in *b* in the output tensor.

y [array_like] Returns a boolean tensor of where *a* and *b* are equal within the given tolerance. If both *a* and *b* are scalars, returns a single boolean value.

allclose

For finite values, isclose uses the following equation to test whether two floating point values are equivalent.

absolute(*a* - *b*) <= (*atol* + *rtol* * absolute(*b*))

The above equation is not symmetric in *a* and *b*, so that *isclose(a, b)* might be different from *isclose(b, a)* in some rare cases.

```
>>> import mars.tensor as mt
```

```
>>> mt.isclose([1e10,1e-7], [1.00001e10,1e-8]).execute()
array([True, False])
>>> mt.isclose([1e10,1e-8], [1.00001e10,1e-9]).execute()
array([True, True])
>>> mt.isclose([1e10,1e-8], [1.0001e10,1e-9]).execute()
array([False, True])
>>> mt.isclose([1.0, mt.nan], [1.0, mt.nan]).execute()
array([True, False])
>>> mt.isclose([1.0, mt.nan], [1.0, mt.nan], equal_nan=True).execute()
array([True, True])
```

mars.tensor.array_equal

mars.tensor.**array_equal**(*a1*, *a2*)

True if two tensors have the same shape and elements, False otherwise.

a1, a2 [array_like] Input arrays.

b [bool] Returns True if the tensors are equal.

allclose: Returns True if two tensors are element-wise equal within a tolerance.

array_equiv: Returns True if input tensors are shape consistent and all elements equal.

```
>>> import mars.tensor as mt
```

```
>>> mt.array_equal([1, 2], [1, 2]).execute()
True
>>> mt.array_equal(mt.array([1, 2]), mt.array([1, 2])).execute()
True
>>> mt.array_equal([1, 2], [1, 2, 3]).execute()
False
>>> mt.array_equal([1, 2], [1, 4]).execute()
False
```

2.6.8 Mathematical Functions

Trigonometric functions

mars.tensor.sin	Trigonometric sine, element-wise.
mars.tensor.cos	Cosine element-wise.
mars.tensor.tan	Compute tangent element-wise.
mars.tensor.arcsin	Inverse sine, element-wise.
mars.tensor.arccos	Trigonometric inverse cosine, element-wise.
mars.tensor.arctan	Trigonometric inverse tangent, element-wise.
mars.tensor.hypot	Given the "legs" of a right triangle, return its hypotenuse.
mars.tensor.arctan2	Element-wise arc tangent of `x1/x2` choosing the quadrant correctly.
mars.tensor.degrees	Convert angles from radians to degrees.
mars.tensor.radians	Convert angles from degrees to radians.
mars.tensor.deg2rad	Convert angles from degrees to radians.
mars.tensor.rad2deg	Convert angles from radians to degrees.

mars.tensor.degrees

mars.tensor.**degrees**(*x*, *out=None*, *where=None*, ***kwargs*)

Convert angles from radians to degrees.

x [array_like] Input tensor in radians.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y [Tensor of floats] The corresponding degree values; if *out* was supplied this is a reference to it.

rad2deg : equivalent function

Convert a radian array to degrees

```
>>> import mars.tensor as mt
```

```
>>> rad = mt.arange(12.)*mt.pi/6
>>> mt.degrees(rad).execute()
```

(continues on next page)

(continued from previous page)

```
array([   0.,   30.,   60.,   90.,  120.,  150.,  180.,  210.,  240.,
        270.,  300.,  330.])
```

```
>>> out = mt.zeros((rad.shape))
>>> r = mt.degrees(out)
>>> mt.all(r == out).execute()
True
```

mars.tensor.radians

mars.tensor.**radians**(*x*, *out=None*, *where=None*, ***kwargs*)

Convert angles from degrees to radians.

x [array_like] Input tensor in degrees.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y [Tensor] The corresponding radian values.

deg2rad : equivalent function

Convert a degree array to radians

```
>>> import mars.tensor as mt
```

```
>>> deg = mt.arange(12.) * 30.
>>> mt.radians(deg).execute()
array([ 0.        ,  0.52359878,  1.04719755,  1.57079633,  2.0943951 ,
        2.61799388,  3.14159265,  3.66519143,  4.1887902 ,  4.71238898,
        5.23598776,  5.75958653])
```

```
>>> out = mt.zeros((deg.shape))
>>> ret = mt.radians(deg, out)
>>> ret is out
True
```

Hyperbolic functions

mars.tensor.sinh	Hyperbolic sine, element-wise.
mars.tensor.cosh	Hyperbolic cosine, element-wise.
mars.tensor.tanh	Compute hyperbolic tangent element-wise.
mars.tensor.arcsinh	Inverse hyperbolic sine element-wise.
mars.tensor.arccosh	Inverse hyperbolic cosine, element-wise.
mars.tensor.arctanh	Inverse hyperbolic tangent element-wise.

Rounding

mars.tensor.around	Evenly round to the given number of decimals.
*mars.tensor.round_*	Evenly round to the given number of decimals.
mars.tensor.rint	Round elements of the tensor to the nearest integer.
mars.tensor.fix	Round to nearest integer towards zero.
mars.tensor.floor	Return the floor of the input, element-wise.
mars.tensor.ceil	Return the ceiling of the input, element-wise.
mars.tensor.trunc	Return the truncated value of the input, element-wise.

mars.tensor.around

mars.tensor.**around**(*a*, *decimals=0*, *out=None*)

Evenly round to the given number of decimals.

a [array_like] Input data.

decimals [int, optional] Number of decimal places to round to (default: 0). If decimals is negative, it specifies the number of positions to the left of the decimal point.

out [Tensor, optional] Alternative output tensor in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

rounded_array [Tensor] An tensor of the same type as *a*, containing the rounded values. Unless *out* was specified, a new tensor is created. A reference to the result is returned.

The real and imaginary parts of complex numbers are rounded separately. The result of rounding a float is a float.

Tensor.round : equivalent method

ceil, fix, floor, rint, trunc

For values exactly halfway between rounded decimal values, NumPy rounds to the nearest even value. Thus 1.5 and 2.5 round to 2.0, -0.5 and 0.5 round to 0.0, etc. Results may also be surprising due to the inexact representation of decimal fractions in the IEEE floating point standard[1] and errors introduced when scaling by powers of ten.

```
>>> import mars.tensor as mt
```

```
>>> mt.around([0.37, 1.64]).execute()
array([ 0.,  2.])
>>> mt.around([0.37, 1.64], decimals=1).execute()
array([ 0.4,  1.6])
>>> mt.around([.5, 1.5, 2.5, 3.5, 4.5]).execute() # rounds to nearest even value
array([ 0.,  2.,  2.,  4.,  4.])
>>> mt.around([1,2,3,11], decimals=1).execute() # tensor of ints is returned
array([ 1,  2,  3, 11])
>>> mt.around([1,2,3,11], decimals=-1).execute()
array([ 0,  0,  0, 10])
```

[1] "Lecture Notes on the Status of IEEE 754", William Kahan, http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF

mars.tensor.round_

mars.tensor.**round_**(*a*, *decimals=0*, *out=None*)

Evenly round to the given number of decimals.

a [array_like] Input data.

decimals [int, optional] Number of decimal places to round to (default: 0). If decimals is negative, it specifies the number of positions to the left of the decimal point.

out [Tensor, optional] Alternative output tensor in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

rounded_array [Tensor] An tensor of the same type as *a*, containing the rounded values. Unless *out* was specified, a new tensor is created. A reference to the result is returned.

The real and imaginary parts of complex numbers are rounded separately. The result of rounding a float is a float.

Tensor.round : equivalent method

ceil, fix, floor, rint, trunc

For values exactly halfway between rounded decimal values, NumPy rounds to the nearest even value. Thus 1.5 and 2.5 round to 2.0, -0.5 and 0.5 round to 0.0, etc. Results may also be surprising due to the inexact representation of decimal fractions in the IEEE floating point standard[1] and errors introduced when scaling by powers of ten.

```
>>> import mars.tensor as mt
```

```
>>> mt.around([0.37, 1.64]).execute()
array([ 0.,  2.])
>>> mt.around([0.37, 1.64], decimals=1).execute()
array([ 0.4,  1.6])
>>> mt.around([.5, 1.5, 2.5, 3.5, 4.5]).execute() # rounds to nearest even value
array([ 0.,  2.,  2.,  4.,  4.])
>>> mt.around([1,2,3,11], decimals=1).execute() # tensor of ints is returned
array([ 1,  2,  3, 11])
>>> mt.around([1,2,3,11], decimals=-1).execute()
array([ 0,  0,  0, 10])
```

mars.tensor.fix

mars.tensor.**fix**(*x*, *out=None*, ***kwargs*)

Round to nearest integer towards zero.

Round a tensor of floats element-wise to nearest integer towards zero. The rounded values are returned as floats.

x [array_like] An tensor of floats to be rounded

out [Tensor, optional] Output tensor

out [Tensor of floats] The array of rounded numbers

trunc, floor, ceil around : Round to given number of decimals

[1] "Lecture Notes on the Status of IEEE 754", William Kahan, http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF

```
>>> import mars.tensor as mt
```

```
>>> mt.fix(3.14).execute()
3.0
>>> mt.fix(3).execute()
3.0
>>> mt.fix([2.1, 2.9, -2.1, -2.9]).execute()
array([ 2.,  2., -2., -2.])
```

Sums, products, differences

mars.tensor.prod	Return the product of tensor elements over a given axis.
mars.tensor.sum	Sum of tensor elements over a given axis.
mars.tensor.nanprod	Return the product of array elements over a given axis treating Not a Numbers (NaNs) as ones.
mars.tensor.nansum	Return the sum of array elements over a given axis treating Not a Numbers (NaNs) as zero.
mars.tensor.cumprod	Return the cumulative product of elements along a given axis.
mars.tensor.cumsum	Return the cumulative sum of the elements along a given axis.
mars.tensor.nancumprod	Return the cumulative product of tensor elements over a given axis treating Not a Numbers (NaNs) as one.
mars.tensor.nancumsum	Return the cumulative sum of tensor elements over a given axis treating Not a Numbers (NaNs) as zero.
mars.tensor.diff	Calculate the n-th discrete difference along the given axis.
mars.tensor.ediff1d	The differences between consecutive elements of a tensor.

mars.tensor.prod

mars.tensor.**prod**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=None*, *combine_size=None*)

Return the product of tensor elements over a given axis.

a [array_like] Input data.

axis [None or int or tuple of ints, optional] Axis or axes along which a product is performed. The default, axis=None, will calculate the product of all the elements in the input tensor. If axis is negative it counts from the last to the first axis.

If axis is a tuple of ints, a product is performed on all of the axes specified in the tuple instead of a single axis or all the axes as before.

dtype [dtype, optional] The type of the returned tensor, as well as of the accumulator in which the elements are multiplied. The dtype of *a* is used by default unless *a* has an integer dtype of less precision than the default platform integer. In that case, if *a* is signed then the platform integer is used while if *a* is unsigned then an unsigned integer of the same precision as the platform integer is used.

out [Tensor, optional] Alternative output tensor in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

keepdims [bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

If the default value is passed, then *keepdims* will not be passed through to the *prod* method of sub-classes of *Tensor*, however any non-default value will be. If the sub-classes *sum* method does not implement *keepdims* any exceptions will be raised.

combine_size: int, optional The number of chunks to combine.

product_along_axis [Tensor, see *dtype* parameter above.] An tensor shaped as *a* but with the specified axis removed. Returns a reference to *out* if specified.

Tensor.prod : equivalent method

Arithmetic is modular when using integer types, and no error is raised on overflow. That means that, on a 32-bit platform:

```
>>> import mars.tensor as mt
```

```
>>> x = mt.array([536870910, 536870910, 536870910, 536870910])
>>> mt.prod(x).execute()  # random
16
```

The product of an empty array is the neutral element 1:

```
>>> mt.prod([]).execute()
1.0
```

By default, calculate the product of all elements:

```
>>> mt.prod([1.,2.]).execute()
2.0
```

Even when the input array is two-dimensional:

```
>>> mt.prod([[1.,2.],[3.,4.]]).execute()
24.0
```

But we can also specify the axis over which to multiply:

```
>>> mt.prod([[1.,2.],[3.,4.]], axis=1).execute()
array([  2.,  12.])
```

If the type of *x* is unsigned, then the output type is the unsigned platform integer:

```
>>> x = mt.array([1, 2, 3], dtype=mt.uint8)
>>> mt.prod(x).dtype == mt.uint
True
```

If *x* is of a signed integer type, then the output type is the default platform integer:

```
>>> x = mt.array([1, 2, 3], dtype=mt.int8)
>>> mt.prod(x).dtype == int
True
```

mars.tensor.sum

mars.tensor.**sum**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=None*, *combine_size=None*)

Sum of tensor elements over a given axis.

a [array_like] Elements to sum.

axis [None or int or tuple of ints, optional] Axis or axes along which a sum is performed. The default, axis=None, will sum all of the elements of the input tensor. If axis is negative it counts from the last to the first axis.

If axis is a tuple of ints, a sum is performed on all of the axes specified in the tuple instead of a single axis or all the axes as before.

dtype [dtype, optional] The type of the returned tensor and of the accumulator in which the elements are summed. The dtype of *a* is used by default unless *a* has an integer dtype of less precision than the default platform integer. In that case, if *a* is signed then the platform integer is used while if *a* is unsigned then an unsigned integer of the same precision as the platform integer is used.

out [Tensor, optional] Alternative output tensor in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

keepdims [bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input tensor.

If the default value is passed, then *keepdims* will not be passed through to the *sum* method of sub-classes of *Tensor*, however any non-default value will be. If the sub-classes *sum* method does not implement *keepdims* any exceptions will be raised.

combine_size: int, optional The number of chunks to combine.

sum_along_axis [Tensor] An array with the same shape as *a*, with the specified axis removed. If *a* is a 0-d tensor, or if *axis* is None, a scalar is returned. If an output array is specified, a reference to *out* is returned.

Tensor.sum : Equivalent method.

cumsum : Cumulative sum of tensor elements.

trapz : Integration of tensor values using the composite trapezoidal rule.

mean, average

Arithmetic is modular when using integer types, and no error is raised on overflow.

The sum of an empty array is the neutral element 0:

```
>>> import mars.tensor as mt
```

```
>>> mt.sum([]).execute()
0.0
```

```
>>> mt.sum([0.5, 1.5]).execute()
2.0
>>> mt.sum([0.5, 0.7, 0.2, 1.5], dtype=mt.int32).execute()
1
>>> mt.sum([[0, 1], [0, 5]]).execute()
6
>>> mt.sum([[0, 1], [0, 5]], axis=0).execute()
array([0, 6])
>>> mt.sum([[0, 1], [0, 5]], axis=1).execute()
array([1, 5])
```

If the accumulator is too small, overflow occurs:

```
>>> mt.ones(128, dtype=mt.int8).sum(dtype=mt.int8).execute()
-128
```

mars.tensor.nanprod

mars.tensor.**nanprod**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=None*, *combine_size=None*)

Return the product of array elements over a given axis treating Not a Numbers (NaNs) as ones.

One is returned for slices that are all-NaN or empty.

a [array_like] Tensor containing numbers whose product is desired. If *a* is not an tensor, a conversion is attempted.

axis [int, optional] Axis along which the product is computed. The default is to compute the product of the flattened tensor.

dtype [data-type, optional] The type of the returned tensor and of the accumulator in which the elements are summed. By default, the dtype of *a* is used. An exception is when *a* has an integer type with less precision than the platform (u)intp. In that case, the default will be either (u)int32 or (u)int64 depending on whether the platform is 32 or 64 bits. For inexact inputs, dtype must be inexact.

out [Tensor, optional] Alternate output tensor in which to place the result. The default is `None`. If provided, it must have the same shape as the expected output, but the type will be cast if necessary. See *doc.ufuncs* for details. The casting of NaN to integer can yield unexpected results.

keepdims [bool, optional] If True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

combine_size: int, optional The number of chunks to combine.

nanprod [Tensor] A new tensor holding the result is returned unless *out* is specified, in which case it is returned.

mt.prod : Product across array propagating NaNs. isnan : Show which elements are NaN.

```
>>> import mars.tensor as mt
```

```
>>> mt.nanprod(1).execute()
1
>>> mt.nanprod([1]).execute()
1
>>> mt.nanprod([1, mt.nan]).execute()
1.0
>>> a = mt.array([[1, 2], [3, mt.nan]])
>>> mt.nanprod(a).execute()
6.0
>>> mt.nanprod(a, axis=0).execute()
array([ 3.,  2.])
```

mars.tensor.nansum

mars.tensor.**nansum**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=None*, *combine_size=None*)

Return the sum of array elements over a given axis treating Not a Numbers (NaNs) as zero.

Zero is returned for slices that are all-NaN or empty.

a [array_like] Tensor containing numbers whose sum is desired. If *a* is not an tensor, a conversion is attempted.

axis [int, optional] Axis along which the sum is computed. The default is to compute the sum of the flattened array.

dtype [data-type, optional] The type of the returned tensor and of the accumulator in which the elements are summed. By default, the dtype of *a* is used. An exception is when *a* has an integer type with less precision than the platform (u)intp. In that case, the default will be either (u)int32 or (u)int64 depending on whether the platform is 32 or 64 bits. For inexact inputs, dtype must be inexact.

out [Tensor, optional] Alternate output tensor in which to place the result. The default is `None`. If provided, it must have the same shape as the expected output, but the type will be cast if necessary. See *doc.ufuncs* for details. The casting of NaN to integer can yield unexpected results.

keepdims [bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *a*.

If the value is anything but the default, then *keepdims* will be passed through to the *mean* or *sum* methods of sub-classes of *Tensor*. If the sub-classes methods does not implement *keepdims* any exceptions will be raised.

combine_size: int, optional The number of chunks to combine.

nansum [Tensor.] A new tensor holding the result is returned unless *out* is specified, in which it is returned. The result has the same size as *a*, and the same shape as *a* if *axis* is not None or *a* is a 1-d array.

mt.sum : Sum across tensor propagating NaNs. isnan : Show which elements are NaN. isfinite: Show which elements are not NaN or +/-inf.

If both positive and negative infinity are present, the sum will be Not A Number (NaN).

```
>>> import mars.tensor as mt
```

```
>>> mt.nansum(1).execute()
1
>>> mt.nansum([1]).execute()
1
>>> mt.nansum([1, mt.nan]).execute()
1.0
>>> a = mt.array([[1, 1], [1, mt.nan]])
>>> mt.nansum(a).execute()
3.0
>>> mt.nansum(a, axis=0).execute()
array([ 2.,  1.])
>>> mt.nansum([1, mt.nan, mt.inf]).execute()
inf
>>> mt.nansum([1, mt.nan, mt.NINF]).execute()
-inf
>>> mt.nansum([1, mt.nan, mt.inf, -mt.inf]).execute()  # both +/- infinity present
nan
```

mars.tensor.cumprod

mars.tensor.**cumprod**(*a*, *axis=None*, *dtype=None*, *out=None*)

Return the cumulative product of elements along a given axis.

a [array_like] Input tensor.

axis [int, optional] Axis along which the cumulative product is computed. By default the input is flattened.

dtype [dtype, optional] Type of the returned tensor, as well as of the accumulator in which the elements are multiplied. If *dtype* is not specified, it defaults to the dtype of *a*, unless *a* has an integer dtype with a

precision less than that of the default platform integer. In that case, the default platform integer is used instead.

out [Tensor, optional] Alternative output tensor in which to place the result. It must have the same shape and buffer length as the expected output but the type of the resulting values will be cast if necessary.

cumprod [Tensor] A new tensor holding the result is returned unless *out* is specified, in which case a reference to out is returned.

numpy.doc.ufuncs : Section "Output arguments"

Arithmetic is modular when using integer types, and no error is raised on overflow.

```
>>> import mars.tensor as mt
```

```
>>> a = mt.array([1,2,3])
>>> mt.cumprod(a).execute() # intermediate results 1, 1*2
...                         # total product 1*2*3 = 6
array([1, 2, 6])
>>> a = mt.array([[1, 2, 3], [4, 5, 6]])
>>> mt.cumprod(a, dtype=float).execute() # specify type of output
array([   1.,    2.,    6.,   24.,  120.,  720.])
```

The cumulative product for each column (i.e., over the rows) of *a*:

```
>>> mt.cumprod(a, axis=0).execute()
array([[ 1,  2,  3],
       [ 4, 10, 18]])
```

The cumulative product for each row (i.e. over the columns) of *a*:

```
>>> mt.cumprod(a,axis=1).execute()
array([[  1,   2,   6],
       [  4,  20, 120]])
```

mars.tensor.cumsum

mars.tensor.**cumsum**(*a*, *axis=None*, *dtype=None*, *out=None*)

Return the cumulative sum of the elements along a given axis.

a [array_like] Input tensor.

axis [int, optional] Axis along which the cumulative sum is computed. The default (None) is to compute the cumsum over the flattened tensor.

dtype [dtype, optional] Type of the returned tensor and of the accumulator in which the elements are summed. If *dtype* is not specified, it defaults to the dtype of *a*, unless *a* has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used.

out [Tensor, optional] Alternative output tensor in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary. See *doc.ufuncs* (Section "Output arguments") for more details.

cumsum_along_axis [Tensor.] A new tensor holding the result is returned unless *out* is specified, in which case a reference to *out* is returned. The result has the same size as *a*, and the same shape as *a* if *axis* is not None or *a* is a 1-d tensor.

sum : Sum tensor elements.

trapz : Integration of tensor values using the composite trapezoidal rule.

diff : Calculate the n-th discrete difference along given axis.

Arithmetic is modular when using integer types, and no error is raised on overflow.

```
>>> import mars.tensor as mt
```

```
>>> a = mt.array([[1,2,3], [4,5,6]])
>>> a.execute()
array([[1, 2, 3],
       [4, 5, 6]])
>>> mt.cumsum(a).execute()
array([ 1,  3,  6, 10, 15, 21])
>>> mt.cumsum(a, dtype=float).execute()     # specifies type of output value(s)
array([  1.,   3.,   6.,  10.,  15.,  21.])
```

```
>>> mt.cumsum(a,axis=0).execute()      # sum over rows for each of the 3 columns
array([[1, 2, 3],
       [5, 7, 9]])
>>> mt.cumsum(a,axis=1).execute()      # sum over columns for each of the 2 rows
array([[ 1,  3,  6],
       [ 4,  9, 15]])
```

mars.tensor.nancumprod

mars.tensor.**nancumprod**(*a*, *axis=None*, *dtype=None*, *out=None*)

Return the cumulative product of tensor elements over a given axis treating Not a Numbers (NaNs) as one. The cumulative product does not change when NaNs are encountered and leading NaNs are replaced by ones.

Ones are returned for slices that are all-NaN or empty.

a [array_like] Input tensor.

axis [int, optional] Axis along which the cumulative product is computed. By default the input is flattened.

dtype [dtype, optional] Type of the returned tensor, as well as of the accumulator in which the elements are multiplied. If *dtype* is not specified, it defaults to the dtype of *a*, unless *a* has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used instead.

out [Tensor, optional] Alternative output tensor in which to place the result. It must have the same shape and buffer length as the expected output but the type of the resulting values will be cast if necessary.

nancumprod [Tensor] A new array holding the result is returned unless *out* is specified, in which case it is returned.

mt.cumprod : Cumulative product across array propagating NaNs. isnan : Show which elements are NaN.

```
>>> import mars.tensor as mt
```

```
>>> mt.nancumprod(1).execute()
array([1])
>>> mt.nancumprod([1]).execute()
array([1])
```

(continues on next page)

(continued from previous page)

```
>>> mt.nancumprod([1, mt.nan]).execute()
array([ 1.,  1.])
>>> a = mt.array([[1, 2], [3, mt.nan]])
>>> mt.nancumprod(a).execute()
array([ 1.,  2.,  6.,  6.])
>>> mt.nancumprod(a, axis=0).execute()
array([[ 1.,  2.],
       [ 3.,  2.]])
>>> mt.nancumprod(a, axis=1).execute()
array([[ 1.,  2.],
       [ 3.,  3.]])
```

mars.tensor.nancumsum

mars.tensor.**nancumsum**(*a*, *axis=None*, *dtype=None*, *out=None*)

Return the cumulative sum of tensor elements over a given axis treating Not a Numbers (NaNs) as zero. The cumulative sum does not change when NaNs are encountered and leading NaNs are replaced by zeros.

Zeros are returned for slices that are all-NaN or empty.

a [array_like] Input tensor.

axis [int, optional] Axis along which the cumulative sum is computed. The default (None) is to compute the cumsum over the flattened tensor.

dtype [dtype, optional] Type of the returned tensor and of the accumulator in which the elements are summed. If *dtype* is not specified, it defaults to the dtype of *a*, unless *a* has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used.

out [Tensor, optional] Alternative output tensor in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary. See *doc.ufuncs* (Section "Output arguments") for more details.

nancumsum [Tensor.] A new tensor holding the result is returned unless *out* is specified, in which it is returned. The result has the same size as *a*, and the same shape as *a* if *axis* is not None or *a* is a 1-d tensor.

numpy.cumsum : Cumulative sum across tensor propagating NaNs. isnan : Show which elements are NaN.

```
>>> import mars.tensor as mt
```

```
>>> mt.nancumsum(1).execute()
array([1])
>>> mt.nancumsum([1]).execute()
array([1])
>>> mt.nancumsum([1, mt.nan]).execute()
array([ 1.,  1.])
>>> a = mt.array([[1, 2], [3, mt.nan]])
>>> mt.nancumsum(a).execute()
array([ 1.,  3.,  6.,  6.])
>>> mt.nancumsum(a, axis=0).execute()
array([[ 1.,  2.],
       [ 4.,  2.]])
>>> mt.nancumsum(a, axis=1).execute()
array([[ 1.,  3.],
       [ 3.,  3.]])
```

mars.tensor.diff

mars.tensor.**diff**(*a*, *n=1*, *axis=-1*)

Calculate the n-th discrete difference along the given axis.

The first difference is given by `out[n] = a[n+1] - a[n]` along the given axis, higher differences are calculated by using *diff* recursively.

a [array_like] Input tensor

n [int, optional] The number of times values are differenced. If zero, the input is returned as-is.

axis [int, optional] The axis along which the difference is taken, default is the last axis.

diff [Tensor] The n-th differences. The shape of the output is the same as *a* except along *axis* where the dimension is smaller by *n*. The type of the output is the same as the type of the difference between any two elements of *a*. This is the same as the type of *a* in most cases. A notable exception is *datetime64*, which results in a *timedelta64* output tensor.

gradient, ediff1d, cumsum

Type is preserved for boolean tensors, so the result will contain *False* when consecutive elements are the same and *True* when they differ.

For unsigned integer tensors, the results will also be unsigned. This should not be surprising, as the result is consistent with calculating the difference directly:

```
>>> import mars.tensor as mt
```

```
>>> u8_arr = mt.array([1, 0], dtype=mt.uint8)
>>> mt.diff(u8_arr).execute()
array([255], dtype=uint8)
>>> (u8_arr[1,...] - u8_arr[0,...]).execute()
255
```

If this is not desirable, then the array should be cast to a larger integer type first:

```
>>> i16_arr = u8_arr.astype(mt.int16)
>>> mt.diff(i16_arr).execute()
array([-1], dtype=int16)
```

```
>>> x = mt.array([1, 2, 4, 7, 0])
>>> mt.diff(x).execute()
array([ 1,  2,  3, -7])
>>> mt.diff(x, n=2).execute()
array([  1,   1, -10])
```

```
>>> x = mt.array([[1, 3, 6, 10], [0, 5, 6, 8]])
>>> mt.diff(x).execute()
array([[2, 3, 4],
       [5, 1, 2]])
>>> mt.diff(x, axis=0).execute()
array([[-1,  2,  0, -2]])
```

```
>>> x = mt.arange('1066-10-13', '1066-10-16', dtype=mt.datetime64)
>>> mt.diff(x).execute()
array([1, 1], dtype='timedelta64[D]')
```

mars.tensor.ediff1d

mars.tensor.**ediff1d**(*a*, *to_end=None*, *to_begin=None*)

The differences between consecutive elements of a tensor.

a [array_like] If necessary, will be flattened before the differences are taken.

to_end [array_like, optional] Number(s) to append at the end of the returned differences.

to_begin [array_like, optional] Number(s) to prepend at the beginning of the returned differences.

ediff1d [Tensor] The differences. Loosely, this is `a.flat[1:] - a.flat[:-1]`.

diff, gradient

```
>>> import mars.tensor as mt
```

```
>>> x = mt.array([1, 2, 4, 7, 0])
>>> mt.ediff1d(x).execute()
array([ 1,  2,  3, -7])
```

```
>>> mt.ediff1d(x, to_begin=-99, to_end=mt.array([88, 99])).execute()
array([-99,   1,   2,   3,  -7,  88,  99])
```

The returned tensor is always 1D.

```
>>> y = [[1, 2, 4], [1, 6, 24]]
>>> mt.ediff1d(y).execute()
array([ 1,  2, -3,  5, 18])
```

Exponential and logarithms

mars.tensor.exp	Calculate the exponential of all elements in the input tensor.
mars.tensor.expm1	Calculate `exp(x) - 1` for all elements in the tensor.
mars.tensor.exp2	Calculate *2**p* for all *p* in the input tensor.
mars.tensor.log	Natural logarithm, element-wise.
mars.tensor.log10	Return the base 10 logarithm of the input tensor, element-wise.
mars.tensor.log2	Base-2 logarithm of *x*.
mars.tensor.log1p	Return the natural logarithm of one plus the input tensor, element-wise.
mars.tensor.logaddexp	Logarithm of the sum of exponentiations of the inputs.
mars.tensor.logaddexp2	Logarithm of the sum of exponentiations of the inputs in base-2.

Other special functions

mars.tensor.i0	Modified Bessel function of the first kind, order 0.
mars.tensor.sinc	Return the sinc function.

mars.tensor.i0

mars.tensor.**i0**(*x*, ***kwargs*)

Modified Bessel function of the first kind, order 0.

Usually denoted I_0. This function does broadcast, but will *not* "up-cast" int dtype arguments unless accompanied by at least one float or complex dtype argument (see Raises below).

x [array_like, dtype float or complex] Argument of the Bessel function.

out [Tensor, shape = x.shape, dtype = x.dtype] The modified Bessel function evaluated at each of the elements of *x*.

TypeError: array cannot be safely cast to required type If argument consists exclusively of int dtypes.

scipy.special.iv, scipy.special.ive

We use the algorithm published by Clenshaw[1] and referenced by Abramowitz and Stegun[2], for which the function domain is partitioned into the two intervals [0,8] and (8,inf), and Chebyshev polynomial expansions are employed in each interval. Relative error on the domain [0,30] using IEEE arithmetic is documented[3] as having a peak of 5.8e-16 with an rms of 1.4e-16 (n = 30000).

```
>>> import mars.tensor as mt
```

```
>>> mt.i0([0.]).execute()
array([1.])
>>> mt.i0([0., 1. + 2j]).execute()
array([ 1.00000000+0.j        ,  0.18785373+0.64616944j])
```

mars.tensor.sinc

mars.tensor.**sinc**(*x*, ***kwargs*)

Return the sinc function.

The sinc function is
$sin($
$pix)/($
$pix)$.

x [Tensor] Tensor (possibly multi-dimensional) of values for which to to calculate `sinc(x)`.

out [Tensor] `sinc(x)`, which has the same shape as the input.

`sinc(0)` is the limit value 1.

The name sinc is short for "sine cardinal" or "sinus cardinalis".

The sinc function is used in various signal processing applications, including in anti-aliasing, in the construction of a Lanczos resampling filter, and in interpolation.

For bandlimited interpolation of discrete-time signals, the ideal interpolation kernel is proportional to the sinc function.

[1] C. W. Clenshaw, "Chebyshev series for mathematical functions", in *National Physical Laboratory Mathematical Tables*, vol. 5, London: Her Majesty's Stationery Office, 1962.

[2] M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions*, 10th printing, New York: Dover, 1964, pp. 379. http://www.math.sfu.ca/~cbm/aands/page_379.htm

[3] http://kobesearch.cpan.org/htdocs/Math-Cephes/Math/Cephes.html

```
>>> import mars.tensor as mt
```

```
>>> x = mt.linspace(-4, 4, 41)
>>> mt.sinc(x).execute()
array([ -3.89804309e-17,  -4.92362781e-02,  -8.40918587e-02,
        -8.90384387e-02,  -5.84680802e-02,   3.89804309e-17,
         6.68206631e-02,   1.16434881e-01,   1.26137788e-01,
         8.50444803e-02,  -3.89804309e-17,  -1.03943254e-01,
        -1.89206682e-01,  -2.16236208e-01,  -1.55914881e-01,
         3.89804309e-17,   2.33872321e-01,   5.04551152e-01,
         7.56826729e-01,   9.35489284e-01,   1.00000000e+00,
         9.35489284e-01,   7.56826729e-01,   5.04551152e-01,
         2.33872321e-01,   3.89804309e-17,  -1.55914881e-01,
        -2.16236208e-01,  -1.89206682e-01,  -1.03943254e-01,
        -3.89804309e-17,   8.50444803e-02,   1.26137788e-01,
         1.16434881e-01,   6.68206631e-02,   3.89804309e-17,
        -5.84680802e-02,  -8.90384387e-02,  -8.40918587e-02,
        -4.92362781e-02,  -3.89804309e-17])
```

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(x.execute(), np.sinc(x).execute())
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.title("Sinc Function")
<matplotlib.text.Text object at 0x...>
>>> plt.ylabel("Amplitude")
<matplotlib.text.Text object at 0x...>
>>> plt.xlabel("X")
<matplotlib.text.Text object at 0x...>
>>> plt.show()
```

Floating point routines

mars.tensor.signbit	Returns element-wise True where signbit is set (less than zero).
mars.tensor.copysign	Change the sign of x1 to that of x2, element-wise.
mars.tensor.frexp	Decompose the elements of x into mantissa and twos exponent.
mars.tensor.ldexp	Returns x1 * 2**x2, element-wise.
mars.tensor.nextafter	Return the next floating-point value after x1 towards x2, element-wise.
mars.tensor.spacing	Return the distance between x and the nearest adjacent number.

mars.tensor.spacing

mars.tensor.**spacing**(*x*, *out=None*, *where=None*, ***kwargs*)

Return the distance between x and the nearest adjacent number.

x [array_like] Values to find the spacing of.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

out [array_like] The spacing of values of *x1*.

It can be considered as a generalization of EPS: `spacing(mt.float64(1)) == mt.finfo(mt.float64).eps`, and there should not be any representable number between `x + spacing(x)` and x for any finite x.

Spacing of +- inf and NaN is NaN.

```
>>> import mars.tensor as mt
```

```
>>> (mt.spacing(1) == mt.finfo(mt.float64).eps).execute()
True
```

Arithmetic operations

mars.tensor.add	Add arguments element-wise.
mars.tensor.reciprocal	Return the reciprocal of the argument, element-wise.
mars.tensor.positive	Numerical positive, element-wise.
mars.tensor.negative	Numerical negative, element-wise.
mars.tensor.multiply	Multiply arguments element-wise.
mars.tensor.divide	Divide arguments element-wise.
mars.tensor.power	First tensor elements raised to powers from second tensor, element-wise.
mars.tensor.subtract	Subtract arguments, element-wise.
mars.tensor.true_divide	Returns a true division of the inputs, element-wise.
mars.tensor.floor_divide	Return the largest integer smaller or equal to the division of the inputs.
mars.tensor.float_power	First tensor elements raised to powers from second array, element-wise.
mars.tensor.fmod	Return the element-wise remainder of division.
mars.tensor.mod	Return element-wise remainder of division.
mars.tensor.modf	Return the fractional and integral parts of a tensor, element-wise.
mars.tensor.remainder	Return element-wise remainder of division.

mars.tensor.positive

mars.tensor.**positive**(*x*, *out=None*, *where=None*, ***kwargs*)

Numerical positive, element-wise.

x [array_like or scalar] Input tensor.

y [Tensor or scalar] Returned array or scalar: $y = +x$.

mars.tensor.float_power

mars.tensor.**float_power**(*x1*, *x2*, *out=None*, *where=None*, ***kwargs*)

First tensor elements raised to powers from second array, element-wise.

Raise each base in *x1* to the positionally-corresponding power in *x2*. *x1* and *x2* must be broadcastable to the same shape. This differs from the power function in that integers, float16, and float32 are promoted to floats with a minimum precision of float64 so that the result is always inexact. The intent is that the function will return a usable result for negative powers and seldom overflow for positive powers.

x1 [array_like] The bases.

x2 [array_like] The exponents.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y [Tensor] The bases in *x1* raised to the exponents in *x2*.

power : power function that preserves type

Cube each element in a list.

```
>>> import mars.tensor as mt
```

```
>>> x1 = range(6)
>>> x1
[0, 1, 2, 3, 4, 5]
>>> mt.float_power(x1, 3).execute()
array([   0.,    1.,    8.,   27.,   64.,  125.])
```

Raise the bases to different exponents.

```
>>> x2 = [1.0, 2.0, 3.0, 3.0, 2.0, 1.0]
>>> mt.float_power(x1, x2).execute()
array([  0.,   1.,   8.,  27.,  16.,   5.])
```

The effect of broadcasting.

```
>>> x2 = mt.array([[1, 2, 3, 3, 2, 1], [1, 2, 3, 3, 2, 1]])
>>> x2.execute()
array([[1, 2, 3, 3, 2, 1],
       [1, 2, 3, 3, 2, 1]])
>>> mt.float_power(x1, x2).execute()
array([[  0.,   1.,   8.,  27.,  16.,   5.],
       [  0.,   1.,   8.,  27.,  16.,   5.]])
```

Handling complex numbers

mars.tensor.angle	Return the angle of the complex argument.
mars.tensor.real	Return the real part of the complex argument.

Continued on next page

Table 43 – continued from previous page

mars.tensor.imag	Return the imaginary part of the complex argument.
mars.tensor.conj	Return the complex conjugate, element-wise.

mars.tensor.angle

mars.tensor.**angle**(*z*, *deg=0*, ***kwargs*)

Return the angle of the complex argument.

z [array_like] A complex number or sequence of complex numbers.

deg [bool, optional] Return angle in degrees if True, radians if False (default).

angle [Tensor or scalar] The counterclockwise angle from the positive real axis on the complex plane, with dtype as numpy.float64.

arctan2 absolute

```
>>> import mars.tensor as mt
```

```
>>> mt.angle([1.0, 1.0j, 1+1j]).execute()               # in radians
array([ 0.        ,  1.57079633,  0.78539816])
>>> mt.angle(1+1j, deg=True).execute()                  # in degrees
45.0
```

mars.tensor.real

mars.tensor.**real**(*val*, ***kwargs*)

Return the real part of the complex argument.

val [array_like] Input tensor.

out [Tensor or scalar] The real component of the complex argument. If *val* is real, the type of *val* is used for the output. If *val* has complex elements, the returned type is float.

real_if_close, imag, angle

```
>>> import mars.tensor as mt
```

```
>>> a = mt.array([1+2j, 3+4j, 5+6j])
>>> a.real.execute()
array([ 1.,  3.,  5.])
>>> a.real = 9
>>> a.execute()
array([ 9.+2.j,  9.+4.j,  9.+6.j])
>>> a.real = mt.array([9, 8, 7])
>>> a.execute()
array([ 9.+2.j,  8.+4.j,  7.+6.j])
>>> mt.real(1 + 1j).execute()
1.0
```

mars.tensor.imag

mars.tensor.**imag**(*val*, ***kwargs*)

Return the imaginary part of the complex argument.

val [array_like] Input tensor.

out [Tensor or scalar] The imaginary component of the complex argument. If *val* is real, the type of *val* is used for the output. If *val* has complex elements, the returned type is float.

real, angle, real_if_close

```
>>> import mars.tensor as mt
```

```
>>> a = mt.array([1+2j, 3+4j, 5+6j])
>>> a.imag.execute()
array([ 2.,  4.,  6.])
>>> a.imag = mt.array([8, 10, 12])
>>> a.execute()
array([ 1. +8.j,  3.+10.j,  5.+12.j])
>>> mt.imag(1 + 1j).execute()
1.0
```

mars.tensor.conj

mars.tensor.**conj**(*x*, *out=None*, *where=None*, ***kwargs*)

Return the complex conjugate, element-wise.

The complex conjugate of a complex number is obtained by changing the sign of its imaginary part.

x [array_like] Input value.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

****kwargs**

y [Tensor] The complex conjugate of *x*, with same dtype as *y*.

```
>>> import mars.tensor as mt
```

```
>>> mt.conjugate(1+2j).execute()
(1-2j)
```

```
>>> x = mt.eye(2) + 1j * mt.eye(2)
>>> mt.conjugate(x).execute()
array([[ 1.-1.j,  0.-0.j],
       [ 0.-0.j,  1.-1.j]])
```

Miscellaneous

mars.tensor.clip	Clip (limit) the values in a tensor.
mars.tensor.sqrt	Return the positive square-root of an tensor, element-wise.
mars.tensor.cbrt	Return the cube-root of an tensor, element-wise.
mars.tensor.square	Return the element-wise square of the input.
mars.tensor.absolute	Calculate the absolute value element-wise.
mars.tensor.sign	Returns an element-wise indication of the sign of a number.
mars.tensor.maximum	Element-wise maximum of tensor elements.
mars.tensor.minimum	Element-wise minimum of tensor elements.
mars.tensor.fmax	Element-wise maximum of array elements.
mars.tensor.fmin	Element-wise minimum of array elements.
mars.tensor.nan_to_num	Replace nan with zero and inf with large finite numbers.

mars.tensor.clip

mars.tensor.**clip**(*a*, *a_min*, *a_max*, *out=None*)

Clip (limit) the values in a tensor.

Given an interval, values outside the interval are clipped to the interval edges. For example, if an interval of `[0, 1]` is specified, values smaller than 0 become 0, and values larger than 1 become 1.

a [array_like] Tensor containing elements to clip.

a_min [scalar or array_like or *None*] Minimum value. If *None*, clipping is not performed on lower interval edge. Not more than one of *a_min* and *a_max* may be *None*.

a_max [scalar or array_like or *None*] Maximum value. If *None*, clipping is not performed on upper interval edge. Not more than one of *a_min* and *a_max* may be *None*. If *a_min* or *a_max* are array_like, then the three arrays will be broadcasted to match their shapes.

out [Tensor, optional] The results will be placed in this tensor. It may be the input array for in-place clipping. *out* must be of the right shape to hold the output. Its type is preserved.

clipped_array [Tensor] An tensor with the elements of *a*, but where values < *a_min* are replaced with *a_min*, and those > *a_max* with *a_max*.

```
>>> import mars.tensor as mt
```

```
>>> a = mt.arange(10)
>>> mt.clip(a, 1, 8).execute()
array([1, 1, 2, 3, 4, 5, 6, 7, 8, 8])
>>> a.execute()
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> mt.clip(a, 3, 6, out=a).execute()
array([3, 3, 3, 3, 4, 5, 6, 6, 6, 6])
>>> a = mt.arange(10)
>>> a.execute()
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> mt.clip(a, [3, 4, 1, 1, 1, 4, 4, 4, 4, 4], 8).execute()
array([3, 4, 2, 3, 4, 5, 6, 7, 8, 8])
```

mars.tensor.cbrt

mars.tensor.**cbrt**(*x*, *out=None*, *where=None*, ***kwargs*)

Return the cube-root of an tensor, element-wise.

x [array_like] The values whose cube-roots are required.

out [Tensor, None, or tuple of Tensor and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or *None*, a freshly-allocated tensor is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where [array_like, optional] Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs

y [Tensor] An tensor of the same shape as *x*, containing the cube cube-root of each element in *x*. If *out* was provided, *y* is a reference to it.

```
>>> import mars.tensor as mt
```

```
>>> mt.cbrt([1,8,27]).execute()
array([ 1.,  2.,  3.])
```

mars.tensor.nan_to_num

mars.tensor.**nan_to_num**(*x*, *copy=True*, ***kwargs*)

Replace nan with zero and inf with large finite numbers.

If *x* is inexact, NaN is replaced by zero, and infinity and -infinity replaced by the respectively largest and most negative finite floating point values representable by `x.dtype`.

For complex dtypes, the above is applied to each of the real and imaginary components of *x* separately.

If *x* is not inexact, then no replacements are made.

x [array_like] Input data.

copy [bool, optional] Whether to create a copy of *x* (True) or to replace values in-place (False). The in-place operation only occurs if casting to an array does not require a copy. Default is True.

out [Tensor] *x*, with the non-finite values replaced. If *copy* is False, this may be *x* itself.

isinf : Shows which elements are positive or negative infinity. isneginf : Shows which elements are negative infinity. isposinf : Shows which elements are positive infinity. isnan : Shows which elements are Not a Number (NaN). isfinite : Shows which elements are finite (not NaN, not infinity)

Mars uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity.

```
>>> import mars.tensor as mt
```

```
>>> x = mt.array([mt.inf, -mt.inf, mt.nan, -128, 128])
>>> mt.nan_to_num(x).execute()
array([  1.79769313e+308,  -1.79769313e+308,   0.00000000e+000,
        -1.28000000e+002,   1.28000000e+002])
>>> y = mt.array([complex(mt.inf, mt.nan), mt.nan, complex(mt.nan, mt.inf)])
```

(continues on next page)

(continued from previous page)

```
>>> mt.nan_to_num(y).execute()
array([  1.79769313e+308 +0.00000000e+000j,
         0.00000000e+000 +0.00000000e+000j,
         0.00000000e+000 +1.79769313e+308j])
```

2.6.9 Random Sampling

Sample random data

mars.tensor.random.rand	Random values in a given shape.
mars.tensor.random.randn	Return a sample (or samples) from the "standard normal" distribution.
mars.tensor.random.randint	Return random integers from *low* (inclusive) to *high* (exclusive).
mars.tensor.random.random_integers	Random integers of type mt.int between *low* and *high*, inclusive.
mars.tensor.random.random_sample	Return random floats in the half-open interval [0.0, 1.0).
mars.tensor.random.random	Return random floats in the half-open interval [0.0, 1.0).
mars.tensor.random.ranf	Return random floats in the half-open interval [0.0, 1.0).
mars.tensor.random.sample	Return random floats in the half-open interval [0.0, 1.0).
mars.tensor.random.choice	Generates a random sample from a given 1-D array
mars.tensor.random.bytes	Return random bytes.

mars.tensor.random.rand

mars.tensor.random.**rand = <bound method rand of <mars.tensor.random.core.RandomState object**

Random values in a given shape.

Create a tensor of the given shape and populate it with random samples from a uniform distributionc over `[0, 1)`.

d0, d1, ..., dn [int, optional] The dimensions of the returned tensor, should all be positive. If no argument is given a single Python float is returned.

out [Tensor, shape `(d0, d1, ..., dn)`] Random values.

random

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to mt.random.random_sample .

```
>>> import mars.tensor as mt
```

```
>>> mt.random.rand(3, 2).execute()
array([[ 0.14022471,  0.96360618],  #random
       [ 0.37601032,  0.25528411],  #random
       [ 0.49313049,  0.94909878]]) #random
```

mars.tensor.random.randn

mars.tensor.random.**randn = <bound method randn of <mars.tensor.random.core.RandomState obj**

Return a sample (or samples) from the "standard normal" distribution.

If positive, int_like or int-convertible arguments are provided, *randn* generates an array of shape `(d0, d1, ..., dn)`, filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.

This is a convenience function. If you want an interface that takes a tuple as the first argument, use *numpy.random.standard_normal* instead.

d0, d1, ..., dn [int, optional] The dimensions of the returned tensor, should be all positive. If no argument is given a single Python float is returned.

Z [Tensor or float] A `(d0, d1, ..., dn)`-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

random.standard_normal : Similar, but takes a tuple as its argument.

For random samples from $N(\mu, \sigma^2)$, use:

`sigma * mt.random.randn(...) + mu`

```
>>> import mars.tensor as mt
```

```
>>> mt.random.randn().execute()
2.1923875335537315 #random
```

Two-by-four tensor of samples from N(3, 6.25):

```
>>> (2.5 * mt.random.randn(2, 4) + 3).execute()
array([[-4.49401501,  4.00950034, -1.81814867,  7.29718677],  #random
       [ 0.39924804,  4.68456316,  4.99394529,  4.84057254]]) #random
```

mars.tensor.random.randint

mars.tensor.random.**randint = <bound method randint of <mars.tensor.random.core.RandomState**

Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the "discrete uniform" distribution of the specified dtype in the "half-open" interval [*low*, *high*). If *high* is None (the default), then results are from [0, *low*).

low [int] Lowest (signed) integer to be drawn from the distribution (unless `high=None`, in which case this parameter is one above the *highest* such integer).

high [int, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`).

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

dtype [dtype, optional] Desired dtype of the result. All dtypes are determined by their name, i.e., 'int64', 'int', etc, so byteorder is not available and a specific precision may have different C types depending on the platform. The default value is 'np.int'.

density: float, optional if density specified, a sparse tensor will be created

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [int or Tensor of ints] *size*-shaped tensor of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random.random_integers [similar to *randint*, only for the closed] interval [*low*, *high*], and 1 is the lowest value if *high* is omitted. In particular, this other one is the one to use to generate uniformly distributed discrete non-integers.

```
>>> import mars.tensor as mt
```

```
>>> mt.random.randint(2, size=10).execute()
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])
>>> mt.random.randint(1, size=10).execute()
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 tensor of ints between 0 and 4, inclusive:

```
>>> mt.random.randint(5, size=(2, 4)).execute()
array([[4, 0, 2, 1],
       [3, 2, 2, 0]])
```

mars.tensor.random.random_integers

mars.tensor.random.**random_integers = <bound method random_integers of <mars.tensor.random.**

Random integers of type mt.int between *low* and *high*, inclusive.

Return random integers of type mt.int from the "discrete uniform" distribution in the closed interval [*low*, *high*]. If *high* is None (the default), then results are from [1, *low*]. The np.int type translates to the C long type used by Python 2 for "short" integers and its precision is platform dependent.

This function has been deprecated. Use randint instead.

low [int] Lowest (signed) integer to be drawn from the distribution (unless `high=None`, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`).

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

out [int or Tensor of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random.randint [Similar to *random_integers*, only for the half-open] interval [*low*, *high*), and 0 is the lowest value if *high* is omitted.

To sample from N evenly spaced floating-point numbers between a and b, use:

```
a + (b - a) * (np.random.random_integers(N) - 1) / (N - 1.)
```

```
>>> import mars.tensor as mt
```

```
>>> mt.random.random_integers(5).execute()
4
>>> type(mt.random.random_integers(5).execute())
<type 'int'>
>>> mt.random.random_integers(5, size=(3,2)).execute()
array([[5, 4],
       [3, 3],
       [4, 5]])
```

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (*i.e.*, from the set $0, 5/8, 10/8, 15/8, 20/8$):

```
>>> (2.5 * (mt.random.random_integers(5, size=(5,)) - 1) / 4.).execute()
array([ 0.625,  1.25 ,  0.625,  0.625,  2.5  ])
```

Roll two six sided dice 1000 times and sum the results:

```
>>> d1 = mt.random.random_integers(1, 6, 1000)
>>> d2 = mt.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums.execute(), 11, normed=True)
>>> plt.show()
```

mars.tensor.random.random_sample

mars.tensor.random.**random_sample = <bound method random_sample of <mars.tensor.random.core**

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of *random_sample* by *(b-a)* and add a:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [float or Tensor of floats] Array of random floats of shape *size* (unless `size=None`, in which case a single float is returned).

```
>>> import mars.tensor as mt
```

```
>>> mt.random.random_sample().execute()
0.47108547995356098
>>> type(mt.random.random_sample().execute())
<type 'float'>
>>> mt.random.random_sample((5,)).execute()
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
>>> (5 * mt.random.random_sample((3, 2)) - 5).execute()
array([[-3.99149989, -0.52338984],
       [-2.99091858, -0.79479508],
       [-1.23204345, -1.75224494]])
```

mars.tensor.random.random

mars.tensor.random.**random = <bound method random_sample of <mars.tensor.random.core.RandomS**

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of *random_sample* by *(b-a)* and add *a*:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [float or Tensor of floats] Array of random floats of shape *size* (unless `size=None`, in which case a single float is returned).

```
>>> import mars.tensor as mt
```

```
>>> mt.random.random_sample().execute()
0.47108547995356098
>>> type(mt.random.random_sample().execute())
<type 'float'>
>>> mt.random.random_sample((5,)).execute()
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
>>> (5 * mt.random.random_sample((3, 2)) - 5).execute()
array([[-3.99149989, -0.52338984],
       [-2.99091858, -0.79479508],
       [-1.23204345, -1.75224494]])
```

mars.tensor.random.ranf

mars.tensor.random.**ranf = <bound method random_sample of <mars.tensor.random.core.RandomSta**

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of *random_sample* by *(b-a)* and add *a*:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [float or Tensor of floats] Array of random floats of shape *size* (unless `size=None`, in which case a single float is returned).

```
>>> import mars.tensor as mt
```

```
>>> mt.random.random_sample().execute()
0.47108547995356098
>>> type(mt.random.random_sample().execute())
<type 'float'>
>>> mt.random.random_sample((5,)).execute()
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
>>> (5 * mt.random.random_sample((3, 2)) - 5).execute()
array([[-3.99149989, -0.52338984],
       [-2.99091858, -0.79479508],
       [-1.23204345, -1.75224494]])
```

mars.tensor.random.sample

mars.tensor.random.**sample = <bound method random_sample of <mars.tensor.random.core.RandomS**

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of *random_sample* by *(b-a)* and add *a*:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [float or Tensor of floats] Array of random floats of shape *size* (unless `size=None`, in which case a single float is returned).

```
>>> import mars.tensor as mt
```

```
>>> mt.random.random_sample().execute()
0.47108547995356098
>>> type(mt.random.random_sample().execute())
<type 'float'>
>>> mt.random.random_sample((5,)).execute()
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
>>> (5 * mt.random.random_sample((3, 2)) - 5).execute()
array([[-3.99149989, -0.52338984],
       [-2.99091858, -0.79479508],
       [-1.23204345, -1.75224494]])
```

mars.tensor.random.choice

mars.tensor.random.**choice = <bound method choice of <mars.tensor.random.core.RandomState ob**

Generates a random sample from a given 1-D array

a [1-D array-like or int] If a tensor, a random sample is generated from its elements. If an int, the random sample is generated as if a were mt.arange(a)

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

replace [boolean, optional] Whether the sample is with or without replacement

p [1-D array-like, optional] The probabilities associated with each entry in a. If not given the sample assumes a uniform distribution over all entries in a.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

samples [single item or tensor] The generated random samples

ValueError If a is an int and less than zero, if a or p are not 1-dimensional, if a is an array-like of size 0, if p is not a vector of probabilities, if a and p have different lengths, or if replace=False and the sample size is greater than the population size

randint, shuffle, permutation

Generate a uniform random sample from mt.arange(5) of size 3:

```
>>> import mars.tensor as mt
```

```
>>> mt.random.choice(5, 3).execute()
array([0, 3, 4])
>>> #This is equivalent to mt.random.randint(0,5,3)
```

Generate a non-uniform random sample from np.arange(5) of size 3:

```
>>> mt.random.choice(5, 3, p=[0.1, 0, 0.3, 0.6, 0]).execute()
array([3, 3, 0])
```

Generate a uniform random sample from mt.arange(5) of size 3 without replacement:

```
>>> mt.random.choice(5, 3, replace=False).execute()
array([3,1,0])
>>> #This is equivalent to np.random.permutation(np.arange(5))[:3]
```

Generate a non-uniform random sample from mt.arange(5) of size 3 without replacement:

```
>>> mt.random.choice(5, 3, replace=False, p=[0.1, 0, 0.3, 0.6, 0]).execute()
array([2, 3, 0])
```

Any of the above can be repeated with an arbitrary array-like instead of just integers. For instance:

```
>>> aa_milne_arr = ['pooh', 'rabbit', 'piglet', 'Christopher']
>>> np.random.choice(aa_milne_arr, 5, p=[0.5, 0.1, 0.1, 0.3])
array(['pooh', 'pooh', 'pooh', 'Christopher', 'piglet'],
      dtype='|S11')
```

mars.tensor.random.bytes

mars.tensor.random.**bytes = <bound method bytes of <mars.tensor.random.core.RandomState obje**

Return random bytes.

length [int] Number of random bytes.

out [str] String of length *length*.

```
>>> import mars.tensor as mt
```

```
>>> mt.random.bytes(10)
' eh
```

2SZ¿¤’ #random

Distributions

mars.tensor.random.beta	Draw samples from a Beta distribution.
mars.tensor.random.binomial	Draw samples from a binomial distribution.
mars.tensor.random.chisquare	Draw samples from a chi-square distribution.
mars.tensor.random.dirichlet	Draw samples from the Dirichlet distribution.
mars.tensor.random.exponential	Draw samples from an exponential distribution.
mars.tensor.random.f	Draw samples from an F distribution.
mars.tensor.random.gamma	Draw samples from a Gamma distribution.
mars.tensor.random.geometric	Draw samples from the geometric distribution.
mars.tensor.random.gumbel	Draw samples from a Gumbel distribution.
mars.tensor.random.hypergeometric	Draw samples from a Hypergeometric distribution.

Continued on next page

Table 46 – continued from previous page

mars.tensor.random.laplace	Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).
mars.tensor.random.lognormal	Draw samples from a log-normal distribution.
mars.tensor.random.logseries	Draw samples from a logarithmic series distribution.
mars.tensor.random.multinomial	Draw samples from a multinomial distribution.
mars.tensor.random.multivariate_normal	Draw random samples from a multivariate normal distribution.
mars.tensor.random.negative_binomial	Draw samples from a negative binomial distribution.
mars.tensor.random.noncentral_chisquare	Draw samples from a noncentral chi-square distribution.
mars.tensor.random.noncentral_f	Draw samples from the noncentral F distribution.
mars.tensor.random.normal	Draw random samples from a normal (Gaussian) distribution.
mars.tensor.random.pareto	Draw samples from a Pareto II or Lomax distribution with specified shape.
mars.tensor.random.poisson	Draw samples from a Poisson distribution.
mars.tensor.random.power	Draws samples in [0, 1] from a power distribution with positive exponent a - 1.
mars.tensor.random.rayleigh	Draw samples from a Rayleigh distribution.
mars.tensor.random.standard_cauchy	Draw samples from a standard Cauchy distribution with mode = 0.
mars.tensor.random.standard_exponential	Draw samples from the standard exponential distribution.
mars.tensor.random.standard_gamma	Draw samples from a standard Gamma distribution.
mars.tensor.random.standard_normal	Draw samples from a standard Normal distribution (mean=0, stdev=1).
mars.tensor.random.standard_t	Draw samples from a standard Student's t distribution with *df* degrees of freedom.
mars.tensor.random.triangular	Draw samples from the triangular distribution over the interval `[left, right]`.
mars.tensor.random.uniform	Draw samples from a uniform distribution.
mars.tensor.random.vonmises	Draw samples from a von Mises distribution.
mars.tensor.random.wald	Draw samples from a Wald, or inverse Gaussian, distribution.
mars.tensor.random.weibull	Draw samples from a Weibull distribution.
mars.tensor.random.zipf	Draw samples from a Zipf distribution.

mars.tensor.random.beta

mars.tensor.random.**beta = <bound method beta of <mars.tensor.random.core.RandomState object**

Draw samples from a Beta distribution.

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

$$f(x; a, b) = \frac{1}{B(\alpha, \beta)} x^{\alpha-1}(1-x)^{\beta-1},$$

where the normalisation, B, is the beta function,

$$B(\alpha, \beta) = \int_0^1 t^{\alpha-1}(1-t)^{\beta-1} dt.$$

It is often seen in Bayesian inference and order statistics.

a [float or array_like of floats] Alpha, non-negative.

b [float or array_like of floats] Beta, non-negative.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` and `b` are both scalars. Otherwise, `mt.broadcast(a, b).size` samples are drawn.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized beta distribution.

mars.tensor.random.binomial

mars.tensor.random.**binomial = <bound method binomial of <mars.tensor.random.core.RandomSta**

Draw samples from a binomial distribution.

Samples are drawn from a binomial distribution with specified parameters, n trials and p probability of success where n an integer >= 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

n [int or array_like of ints] Parameter of the distribution, >= 0. Floats are also accepted, but they will be truncated to integers.

p [float or array_like of floats] Parameter of the distribution, >= 0 and <=1.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `n` and `p` are both scalars. Otherwise, `mt.broadcast(n, p).size` samples are drawn.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized binomial distribution, where each sample is equal to the number of successes over the n trials.

scipy.stats.binom [probability density function, distribution or] cumulative density function, etc.

The probability density for the binomial distribution is

$$P(N) = \binom{n}{N} p^N (1-p)^{n-N},$$

where n is the number of trials, p is the probability of success, and N is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product p*n <=5, where p = population proportion estimate, and n = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4, so the binomial distribution should be used in this case.

Draw samples from the distribution:

```
>>> import mars.tensor as mt
```

```
>>> n, p = 10, .5  # number of trials, probability of each trial
>>> s = mt.random.binomial(n, p, 1000).execute()
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

```
>>> (mt.sum(mt.random.binomial(9, 0.1, 20000) == 0)/20000.).execute()
# answer = 0.38885, or 38%.
```

mars.tensor.random.chisquare

mars.tensor.random.**chisquare = <bound method chisquare of <mars.tensor.random.core.RandomSt**

Draw samples from a chi-square distribution.

When *df* independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

df [float or array_like of floats] Number of degrees of freedom, should be > 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `df` is a scalar. Otherwise, `mt.array(df).size` samples are drawn.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized chi-square distribution.

ValueError When *df* <= 0 or when an inappropriate *size* (e.g. `size=-1`) is given.

The variable obtained by summing the squares of *df* independent, standard normally distributed random variables:

$$Q = \sum_{i=0}^{\mathtt{df}} X_i^2$$

is chi-square distributed, denoted

$$Q \sim \chi_k^2.$$

The probability density function of the chi-squared distribution is

$$p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2-1} e^{-x/2},$$

where Γ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt.$$

```
>>> import mars.tensor as mt
```

```
>>> mt.random.chisquare(2,4).execute()
array([ 1.89920014,  9.00867716,  3.13710533,  5.62318272])
```

mars.tensor.random.dirichlet

mars.tensor.random.**dirichlet = <bound method dirichlet of <mars.tensor.random.core.RandomSt**

Draw samples from the Dirichlet distribution.

Draw *size* samples of dimension k from a Dirichlet distribution. A Dirichlet-distributed random variable can be seen as a multivariate generalization of a Beta distribution. Dirichlet pdf is the conjugate prior of a multinomial in Bayesian inference.

alpha [array] Parameter of the distribution (k dimension for sample of dimension k).

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

samples [Tensor] The drawn samples, of shape (size, alpha.ndim).

ValueError If any value in alpha is less than or equal to zero

$$X \approx \prod_{i=1}^{k} x_i^{\alpha_i - 1}$$

Uses the following property for computation: for each dimension, draw a random sample y_i from a standard gamma generator of shape *alpha_i*, then $X = \frac{1}{\sum_{i=1}^{k} y_i}(y_1, \ldots, y_n)$ is Dirichlet distributed.

Taking an example cited in Wikipedia, this distribution can be used if one wanted to cut strings (each of initial length 1.0) into K pieces with different lengths, where each piece had, on average, a designated average length, but allowing some variation in the relative sizes of the pieces.

```
>>> import mars.tensor as mt
```

```
>>> s = mt.random.dirichlet((10, 5, 3), 20).transpose()
```

```
>>> import matplotlib.pyplot as plt
```

```
>>> plt.barh(range(20), s[0].execute())
>>> plt.barh(range(20), s[1].execute(), left=s[0].execute(), color='g')
>>> plt.barh(range(20), s[2].execute(), left=(s[0]+s[1]).execute(), color='r')
>>> plt.title("Lengths of Strings")
```

mars.tensor.random.exponential

mars.tensor.random.**exponential = <bound method exponential of <mars.tensor.random.core.Rand**

Draw samples from an exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for `x > 0` and 0 elsewhere. β is the scale parameter, which is the inverse of the rate parameter $\lambda = 1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution[3].

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms[1], or the time between page requests to Wikipedia[2].

scale [float or array_like of floats] The scale parameter, $\beta = 1/\lambda$.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `scale` is a scalar. Otherwise, `np.array(scale).size` samples are drawn.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized exponential distribution.

mars.tensor.random.f

mars.tensor.random.**f = <bound method f of <mars.tensor.random.core.RandomState object>>**

Draw samples from an F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters should be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

dfnum [float or array_like of floats] Degrees of freedom in numerator, should be > 0.

dfden [float or array_like of float] Degrees of freedom in denominator, should be > 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `dfnum` and `dfden` are both scalars. Otherwise, `np.broadcast(dfnum, dfden).size` samples are drawn.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized Fisher distribution.

scipy.stats.f [probability density function, distribution or] cumulative density function, etc.

[3] Wikipedia, "Exponential distribution", http://en.wikipedia.org/wiki/Exponential_distribution
[1] Peyton Z. Peebles Jr., "Probability, Random Variables and Random Signal Principles", 4th ed, 2001, p. 57.
[2] Wikipedia, "Poisson process", http://en.wikipedia.org/wiki/Poisson_process

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable *dfnum* is the number of samples minus one, the between-groups degrees of freedom, while *dfden* is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

An example from Glantz[1], pp 47-40:

Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children's blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

```
>>> import mars.tensor as mt
```

```
>>> dfnum = 1. # between group degrees of freedom
>>> dfden = 48. # within groups degrees of freedom
>>> s = mt.random.f(dfnum, dfden, 1000).execute()
```

The lower bound for the top 1% of the samples is :

```
>>> sorted(s)[-10]
7.61988120985
```

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

mars.tensor.random.gamma

mars.tensor.random.**gamma = <bound method gamma of <mars.tensor.random.core.RandomState obj**

Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, *shape* (sometimes designated "k") and *scale* (sometimes designated "theta"), where both parameters are > 0.

shape [float or array_like of floats] The shape of the gamma distribution. Should be greater than zero.

scale [float or array_like of floats, optional] The scale of the gamma distribution. Should be greater than zero. Default is equal to 1.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `shape` and `scale` are both scalars. Otherwise, `np.broadcast(shape, scale).size` samples are drawn.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized gamma distribution.

scipy.stats.gamma [probability density function, distribution or] cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1}\frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> import mars.tensor as mt
```

```
>>> shape, scale = 2., 2.  # mean=4, std=2*sqrt(2)
>>> s = mt.random.gamma(shape, scale, 1000).execute()
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> import numpy as np
>>> count, bins, ignored = plt.hist(s, 50, normed=True)
>>> y = bins**(shape-1)*(np.exp(-bins/scale) /
...                      (sps.gamma(shape)*scale**shape))
>>> plt.plot(bins, y, linewidth=2, color='r')
>>> plt.show()
```

mars.tensor.random.geometric

mars.tensor.random.**geometric = <bound method geometric of <mars.tensor.random.core.RandomSt**

Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, `k = 1, 2, ...`.

The probability mass function of the geometric distribution is

$$f(k) = (1-p)^{k-1}p$$

where *p* is the probability of success of an individual trial.

p [float or array_like of floats] The probability of success of an individual trial.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `p` is a scalar. Otherwise, `mt.array(p).size` samples are drawn.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized geometric distribution.

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```
>>> import mars.tensor as mt
```

```
>>> z = mt.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```
>>> ((z == 1).sum() / 10000.).execute()
0.34889999999999999 #random
```

mars.tensor.random.gumbel

mars.tensor.random.**gumbel = <bound method gumbel of <mars.tensor.random.core.RandomState ob**

Draw samples from a Gumbel distribution.

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

loc [float or array_like of floats, optional] The location of the mode of the distribution. Default is 0.

scale [float or array_like of floats, optional] The scale parameter of the distribution. Default is 1.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `np.broadcast(loc, scale).size` samples are drawn.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized Gumbel distribution.

scipy.stats.gumbel_l scipy.stats.gumbel_r scipy.stats.genextreme weibull

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with "exponential-like" tails.

The probability density for the Gumbel distribution is

$$p(x) = \frac{e^{-(x-\mu)/\beta}}{\beta} e^{-e^{-(x-\mu)/\beta}},$$

where μ is the mode, a location parameter, and β is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a "fat-tailed" distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

Draw samples from the distribution:

```
>>> import mars.tensor as mt
```

```
>>> mu, beta = 0, 0.1 # location and scale
>>> s = mt.random.gumbel(mu, beta, 1000).execute()
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> count, bins, ignored = plt.hist(s, 30, normed=True)
>>> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta)
...          * np.exp( -np.exp( -(bins - mu) /beta) ),
...          linewidth=2, color='r')
>>> plt.show()
```

Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

```
>>> means = []
>>> maxima = []
>>> for i in range(0,1000) :
...    a = mt.random.normal(mu, beta, 1000)
...    means.append(a.mean().execute())
...    maxima.append(a.max().execute())
>>> count, bins, ignored = plt.hist(maxima, 30, normed=True)
>>> beta = mt.std(maxima) * mt.sqrt(6) / mt.pi
>>> mu = mt.mean(maxima) - 0.57721*beta
>>> plt.plot(bins, ((1/beta)*mt.exp(-(bins - mu)/beta)
...          * mt.exp(-mt.exp(-(bins - mu)/beta))).execute(),
...          linewidth=2, color='r')
>>> plt.plot(bins, (1/(beta * mt.sqrt(2 * mt.pi))
...          * mt.exp(-(bins - mu)**2 / (2 * beta**2))).execute(),
...          linewidth=2, color='g')
>>> plt.show()
```

mars.tensor.random.hypergeometric

mars.tensor.random.**hypergeometric = <bound method hypergeometric of <mars.tensor.random.co**

Draw samples from a Hypergeometric distribution.

Samples are drawn from a hypergeometric distribution with specified parameters, ngood (ways to make a good selection), nbad (ways to make a bad selection), and nsample = number of items sampled, which is less than or equal to the sum ngood + nbad.

ngood [int or array_like of ints] Number of ways to make a good selection. Must be nonnegative.

nbad [int or array_like of ints] Number of ways to make a bad selection. Must be nonnegative.

nsample [int or array_like of ints] Number of items sampled. Must be at least 1 and at most `ngood + nbad`.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `ngood`, `nbad`, and `nsample` are all scalars. Otherwise, `np.broadcast(ngood, nbad, nsample).size` samples are drawn.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized hypergeometric distribution.

scipy.stats.hypergeom [probability density function, distribution or] cumulative density function, etc.

The probability density for the Hypergeometric distribution is

$$P(x) = \frac{\binom{m}{n}\binom{N-m}{n-x}}{\binom{N}{n}},$$

where $0 \leq x \leq m$ and $n + m - N \leq x \leq n$

for P(x) the probability of x successes, n = ngood, m = nbad, and N = number of samples.

Consider an urn with black and white marbles in it, ngood of them black and nbad are white. If you draw nsample balls without replacement, then the hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the binomial.

Draw samples from the distribution:

```
>>> import mars.tensor as mt
```

```
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = mt.random.hypergeometric(ngood, nbad, nsamp, 1000)
>>> hist(s)
#   note that it is very unlikely to grab both bad items
```

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```
>>> s = mt.random.hypergeometric(15, 15, 15, 100000)
>>> (mt.sum(s>=12)/100000. + mt.sum(s<=3)/100000.).execute()
#   answer = 0.003 ... pretty unlikely!
```

mars.tensor.random.laplace

mars.tensor.random.**laplace = <bound method laplace of <mars.tensor.random.core.RandomState**

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

loc [float or array_like of floats, optional] The position, μ, of the distribution peak. Default is 0.

scale [float or array_like of floats, optional] λ, the exponential decay. Default is 1.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `np.broadcast(loc, scale).size` samples are drawn.

chunks [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized Laplace distribution.

It has the probability density function

$$f(x; \mu, \lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|x - \mu|}{\lambda}\right).$$

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in economics and health sciences, this distribution seems to model the data better than the standard Gaussian distribution.

Draw samples from the distribution

```
>>> import mars.tensor as mt
```

```
>>> loc, scale = 0., 1.
>>> s = mt.random.laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s.execute(), 30, normed=True)
>>> x = mt.arange(-8., 8., .01)
>>> pdf = mt.exp(-abs(x-loc)/scale)/(2.*scale)
>>> plt.plot(x.execute(), pdf.execute())
```

Plot Gaussian for comparison:

```
>>> g = (1/(scale * mt.sqrt(2 * np.pi)) *
...       mt.exp(-(x - loc)**2 / (2 * scale**2)))
>>> plt.plot(x.execute(),g.execute())
```

mars.tensor.random.lognormal

mars.tensor.random.**lognormal = <bound method lognormal of <mars.tensor.random.core.RandomSt**

Draw samples from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

mean [float or array_like of floats, optional] Mean value of the underlying normal distribution. Default is 0.

sigma [float or array_like of floats, optional] Standard deviation of the underlying normal distribution. Should be greater than zero. Default is 1.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `mean` and `sigma` are both scalars. Otherwise, `np.broadcast(mean, sigma).size` samples are drawn.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized log-normal distribution.

scipy.stats.lognorm [probability density function, distribution,] cumulative density function, etc.

A variable *x* has a log-normal distribution if *log(x)* is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x\sqrt{2\pi}} e^{(-\frac{(ln(x)-\mu)^2}{2\sigma^2})}$$

where μ is the mean and σ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the *product* of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the *sum* of a large number of independent, identically-distributed variables.

Draw samples from the distribution:

```
>>> import mars.tensor as mt
```

```
>>> mu, sigma = 3., 1. # mean and standard deviation
>>> s = mt.random.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s.execute(), 100, normed=True, align='mid')
```

```
>>> x = mt.linspace(min(bins), max(bins), 10000)
>>> pdf = (mt.exp(-(mt.log(x) - mu)**2 / (2 * sigma**2))
...        / (x * sigma * mt.sqrt(2 * mt.pi)))
```

```
>>> plt.plot(x.execute(), pdf.execute(), linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()
```

Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a log-normal probability density function.

```
>>> # Generate a thousand samples: each is the product of 100 random
>>> # values, drawn from a normal distribution.
>>> b = []
>>> for i in range(1000):
...     a = 10. + mt.random.random(100)
...     b.append(mt.product(a).execute())
```

```
>>> b = mt.array(b) / mt.min(b) # scale values to be positive
>>> count, bins, ignored = plt.hist(b.execute(), 100, normed=True, align='mid')
>>> sigma = mt.std(mt.log(b))
>>> mu = mt.mean(mt.log(b))
```

```
>>> x = mt.linspace(min(bins), max(bins), 10000)
>>> pdf = (mt.exp(-(mt.log(x) - mu)**2 / (2 * sigma**2))
...        / (x * sigma * mt.sqrt(2 * mt.pi)))
```

```
>>> plt.plot(x.execute(), pdf.execute(), color='r', linewidth=2)
>>> plt.show()
```

mars.tensor.random.logseries

mars.tensor.random.**logseries = <bound method logseries of <mars.tensor.random.core.RandomSt**

Draw samples from a logarithmic series distribution.

Samples are drawn from a log series distribution with specified shape parameter, 0 < p < 1.

p [float or array_like of floats] Shape parameter for the distribution. Must be in the range (0, 1).

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `p` is a scalar. Otherwise, `np.array(p).size` samples are drawn.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized logarithmic series distribution.

scipy.stats.logser [probability density function, distribution or] cumulative density function, etc.

The probability density for the Log Series distribution is

$$P(k) = \frac{-p^k}{k \ln(1-p)},$$

where p = probability.

The log series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

Draw samples from the distribution:

```
>>> import mars.tensor as mt
>>> import matplotlib.pyplot as plt
```

```
>>> a = .6
>>> s = mt.random.logseries(a, 10000)
>>> count, bins, ignored = plt.hist(s.execute())
```

plot against distribution

```
>>> def logseries(k, p):
...     return -p**k/(k*mt.log(1-p))
>>> plt.plot(bins, (logseries(bins, a)*count.max()/
...          logseries(bins, a).max()).execute(), 'r')
>>> plt.show()
```

mars.tensor.random.multinomial

mars.tensor.random.**multinomial = <bound method multinomial of <mars.tensor.random.core.Rand**

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalisation of the binomial distribution. Take an experiment with one of `p` possible outcomes. An example of such an experiment is throwing a dice, where the outcome

can be 1 through 6. Each sample drawn from the distribution represents *n* such experiments. Its values, `X_i = [X_0, X_1, ..., X_p]`, represent the number of times the outcome was `i`.

n [int] Number of experiments.

pvals [sequence of floats, length p] Probabilities of each of the `p` different outcomes. These should sum to 1 (however, the last element is always assumed to account for the remaining probability, as long as `sum(pvals[:-1]) <= 1)`.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor] The drawn samples, of shape *size*, if that was provided. If not, the shape is `(N,)`.

In other words, each entry `out[i,j,...,:]` is an N-dimensional value drawn from the distribution.

Throw a dice 20 times:

```
>>> import mars.tensor as mt
```

```
>>> mt.random.multinomial(20, [1/6.]*6, size=1).execute()
array([[4, 1, 7, 5, 2, 1]])
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

```
>>> mt.random.multinomial(20, [1/6.]*6, size=2).execute()
array([[3, 4, 3, 3, 4, 3],
       [2, 4, 3, 4, 0, 7]])
```

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded die is more likely to land on number 6:

```
>>> mt.random.multinomial(100, [1/7.]*5 + [2/7.]).execute()
array([11, 16, 14, 17, 16, 26])
```

The probability inputs should be normalized. As an implementation detail, the value of the last entry is ignored and assumed to take up any leftover probability mass, but this should not be relied on. A biased coin which has twice as much weight on one side as on the other should be sampled like so:

```
>>> mt.random.multinomial(100, [1.0 / 3, 2.0 / 3]).execute()  # RIGHT
array([38, 62])
```

not like:

```
>>> mt.random.multinomial(100, [1.0, 2.0]).execute()  # WRONG
array([100,   0])
```

mars.tensor.random.multivariate_normal

mars.tensor.random.**multivariate_normal = <bound method multivariate_normal of <mars.tensor**

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

mean [1-D array_like, of length N] Mean of the N-dimensional distribution.

cov [2-D array_like, of shape (N, N)] Covariance matrix of the distribution. It must be symmetric and positive-semidefinite for proper sampling.

size [int or tuple of ints, optional] Given a shape of, for example, `(m,n,k)`, `m*n*k` samples are generated, and packed in an *m*-by-*n*-by-*k* arrangement. Because each sample is *N*-dimensional, the output shape is `(m,n,k,N)`. If no shape is specified, a single (*N*-D) sample is returned.

check_valid [{ 'warn', 'raise', 'ignore' }, optional] Behavior when the covariance matrix is not positive semidefinite.

tol [float, optional] Tolerance when checking the singular values in covariance matrix.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor] The drawn samples, of shape *size*, if that was provided. If not, the shape is `(N,)`.

In other words, each entry `out[i,j,...,:]` is an N-dimensional value drawn from the distribution.

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ...x_N]$. The covariance matrix element C_{ij} is the covariance of x_i and x_j. The element C_{ii} is the variance of x_i (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- Spherical covariance (*cov* is a multiple of the identity matrix)
- Diagonal covariance (*cov* has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0, 0]
>>> cov = [[1, 0], [0, 100]]  # diagonal covariance
```

Diagonal covariance means that points are oriented along x or y-axis:

```
>>> import matplotlib.pyplot as plt
>>> import mars.tensor as mt
>>> x, y = mt.random.multivariate_normal(mean, cov, 5000).T
>>> plt.plot(x.execute(), y.execute(), 'x')
>>> plt.axis('equal')
>>> plt.show()
```

Note that the covariance matrix must be positive semidefinite (a.k.a. nonnegative-definite). Otherwise, the behavior of this method is undefined and backwards compatibility is not guaranteed.

```
>>> mean = (1, 2)
>>> cov = [[1, 0], [0, 1]]
>>> x = mt.random.multivariate_normal(mean, cov, (3, 3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> list(((x[0,0,:] - mean) < 0.6).execute())
[True, True]
```

mars.tensor.random.negative_binomial

mars.tensor.random.**negative_binomial = <bound method negative_binomial of <mars.tensor.rand**

Draw samples from a negative binomial distribution.

Samples are drawn from a negative binomial distribution with specified parameters, *n* trials and *p* probability of success where *n* is an integer > 0 and *p* is in the interval [0, 1].

n [int or array_like of ints] Parameter of the distribution, > 0. Floats are also accepted, but they will be truncated to integers.

p [float or array_like of floats] Parameter of the distribution, >= 0 and <=1.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `n` and `p` are both scalars. Otherwise, `np.broadcast(n, p).size` samples are drawn.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized negative binomial distribution, where each sample is equal to N, the number of trials it took to achieve n - 1 successes, N - (n - 1) failures, and a success on the, (N + n)th trial.

The probability density for the negative binomial distribution is

$$P(N;n,p) = \binom{N+n-1}{n-1} p^n (1-p)^N,$$

where $n-1$ is the number of successes, p is the probability of success, and $N+n-1$ is the number of trials. The negative binomial distribution gives the probability of n-1 successes and N failures in N+n-1 trials, and success on the (N+n)th trial.

If one throws a die repeatedly until the third time a “1” appears, then the probability distribution of the number of non-“1”s that appear before the third “1” is a negative binomial distribution.

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```
>>> import mars.tensor as mt
```

```
>>> s = mt.random.negative_binomial(1, 0.1, 100000)
>>> for i in range(1, 11):
...     probability = (mt.sum(s<i) / 100000.).execute()
...     print i, "wells drilled, probability of one success =", probability
```

mars.tensor.random.noncentral_chisquare

mars.tensor.random.**noncentral_chisquare = <bound method noncentral_chisquare of <mars.tenso**

Draw samples from a noncentral chi-square distribution.

The noncentral χ^2 distribution is a generalisation of the χ^2 distribution.

df [float or array_like of floats] Degrees of freedom, should be > 0.

nonc [float or array_like of floats] Non-centrality, should be non-negative.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `df` and `nonc` are both scalars. Otherwise, `mt.broadcast(df, nonc).size` samples are drawn.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized noncentral chi-square distribution.

The probability density function for the noncentral Chi-square distribution is

$$P(x;df,nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2}(nonc/2)^i}{i!} \P_{Y_{df+2i}}(x),$$

where Y_q is the Chi-square with q degrees of freedom.

In Delhi (2007), it is noted that the noncentral chi-square is useful in bombing and coverage problems, the probability of killing the point target given by the noncentral chi-squared distribution.

Draw values from the distribution and plot the histogram

```
>>> import matplotlib.pyplot as plt
>>> import mars.tensor as mt
>>> values = plt.hist(mt.random.noncentral_chisquare(3, 20, 100000).execute(),
...                   bins=200, normed=True)
>>> plt.show()
```

Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.

```
>>> plt.figure()
>>> values = plt.hist(mt.random.noncentral_chisquare(3, .0000001, 100000).
↪execute(),
...                   bins=mt.arange(0., 25, .1).execute(), normed=True)
>>> values2 = plt.hist(mt.random.chisquare(3, 100000).execute(),
...                    bins=mt.arange(0., 25, .1).execute(), normed=True)
>>> plt.plot(values[1][0:-1], values[0]-values2[0], 'ob')
>>> plt.show()
```

Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```
>>> plt.figure()
>>> values = plt.hist(mt.random.noncentral_chisquare(3, 20, 100000).execute(),
...                   bins=200, normed=True)
>>> plt.show()
```

mars.tensor.random.noncentral_f

mars.tensor.random.**noncentral_f = <bound method noncentral_f of <mars.tensor.random.core.Ra**

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters > 1. *nonc* is the non-centrality parameter.

dfnum [float or array_like of floats] Numerator degrees of freedom, should be > 0.

dfden [float or array_like of floats] Denominator degrees of freedom, should be > 0.

nonc [float or array_like of floats] Non-centrality parameter, the sum of the squares of the numerator means, should be >= 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `dfnum`, `dfden`, and `nonc` are all scalars. Otherwise, `np.broadcast(dfnum, dfden, nonc).size` samples are drawn.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized noncentral Fisher distribution.

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We'll plot the two probability distributions for comparison.

```
>>> import mars.tensor as mt
>>> import matplotlib.pyplot as plt
```

```
>>> dfnum = 3 # between group deg of freedom
>>> dfden = 20 # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = mt.random.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals.execute(), bins=50, normed=True)  # TODO(jisheng):␣
↪implement mt.histogram
>>> c_vals = mt.random.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals.execute(), bins=50, normed=True)
>>> plt.plot(F[1][1:], F[0])
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```

mars.tensor.random.normal

mars.tensor.random.**normal = <bound method normal of <mars.tensor.random.core.RandomState ob**

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently[2], is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution[2].

loc [float or array_like of floats] Mean ("centre") of the distribution.

scale [float or array_like of floats] Standard deviation (spread or "width") of the distribution.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `mt.broadcast(loc, scale).size` samples are drawn.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized normal distribution.

scipy.stats.norm [probability density function, distribution or] cumulative density function, etc.

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2, is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma^2$). This implies that *numpy.random.normal* is more likely to return samples lying close to the mean, rather than those far away.

Draw samples from the distribution:

```
>>> import mars.tensor as mt
```

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = mt.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```
>>> (abs(mu - mt.mean(s)) < 0.01).execute()
True
```

```
>>> (abs(sigma - mt.std(s, ddof=1)) < 0.01).execute()
True
```

Display the histogram of the samples, along with the probability density function:

[2] P. R. Peebles Jr., "Central Limit Theorem" in "Probability, Random Variables and Random Signal Principles", 4th ed., 2001, pp. 51, 51, 125.

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s.execute(), 30, normed=True)
>>> plt.plot(bins, (1/(sigma * mt.sqrt(2 * mt.pi)) *
...                 mt.exp( - (bins - mu)**2 / (2 * sigma**2) )).execute(),
...          linewidth=2, color='r')
>>> plt.show()
```

mars.tensor.random.pareto

mars.tensor.random.**pareto = <bound method pareto of <mars.tensor.random.core.RandomState ob**

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding 1 and multiplying by the scale parameter `m` (see Notes). The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is `mu`, where the standard Pareto distribution has location `mu = 1`. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the "80-20 rule". In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

a [float or array_like of floats] Shape of the distribution. Should be greater than zero.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `mt.array(a).size` samples are drawn.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized Pareto distribution.

scipy.stats.lomax [probability density function, distribution or] cumulative density function, etc.

scipy.stats.genpareto [probability density function, distribution or] cumulative density function, etc.

The probability density for the Pareto distribution is

$$p(x) = \frac{am^a}{x^{a+1}}$$

where a is the shape and m the scale.

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge[1]. It is one of the so-called "fat-tailed" distributions.

Draw samples from the distribution:

```
>>> import mars.tensor as mt
```

[1] Francis Hunt and Paul Johnson, On the Pareto Distribution of Sourceforge projects.

```
>>> a, m = 3., 2.  # shape and mode
>>> s = (mt.random.pareto(a, 1000) + 1) * m
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, _ = plt.hist(s.execute(), 100, normed=True)
>>> fit = a*m**a / bins**(a+1)
>>> plt.plot(bins, max(count)*fit/max(fit), linewidth=2, color='r')
>>> plt.show()
```

mars.tensor.random.poisson

mars.tensor.random.**poisson = <bound method poisson of <mars.tensor.random.core.RandomState**

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the binomial distribution for large N.

lam [float or array_like of floats] Expectation of interval, should be >= 0. A sequence of expectation intervals must be broadcastable over the requested size.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `lam` is a scalar. Otherwise, `mt.array(lam).size` samples are drawn.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized Poisson distribution.

The Poisson distribution

$$f(k; \lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation λ the Poisson distribution $f(k; \lambda)$ describes the probability of k events occurring within the observed interval λ.

Because the output is limited to the range of the C long type, a ValueError is raised when *lam* is within 10 sigma of the maximum representable value.

Draw samples from the distribution:

```
>>> import mars.tensor as mt
>>> s = mt.random.poisson(5, 10000)
```

Display histogram of the sample:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s.execute(), 14, normed=True)
>>> plt.show()
```

Draw each 100 values for lambda 100 and 500:

```
>>> s = mt.random.poisson(lam=(100., 500.), size=(100, 2))
```

mars.tensor.random.power

mars.tensor.random.**power = <bound method power of <mars.tensor.random.core.RandomState obj**

Draws samples in [0, 1] from a power distribution with positive exponent a - 1.

Also known as the power function distribution.

a [float or array_like of floats] Parameter of the distribution. Should be greater than zero.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `mt.array(a).size` samples are drawn.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized power distribution.

ValueError If a < 1.

The probability density function is

$$P(x; a) = ax^{a-1}, 0 \leq x \leq 1, a > 0.$$

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

Draw samples from the distribution:

```
>>> import mars.tensor as mt
```

```
>>> a = 5. # shape
>>> samples = 1000
>>> s = mt.random.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s.execute(), bins=30)
>>> x = mt.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*mt.diff(bins)[0]*y
>>> plt.plot(x.execute(), normed_y.execute())
>>> plt.show()
```

Compare the power function distribution to the inverse of the Pareto.

```
>>> from scipy import stats
>>> rvs = mt.random.power(5, 1000000)
>>> rvsp = mt.random.pareto(5, 1000000)
>>> xx = mt.linspace(0,1,100)
>>> powpdf = stats.powerlaw.pdf(xx.execute(),5)
```

```
>>> plt.figure()
>>> plt.hist(rvs.execute(), bins=50, normed=True)
>>> plt.plot(xx.execute(),powpdf,'r-')
>>> plt.title('np.random.power(5)')
```

```
>>> plt.figure()
>>> plt.hist((1./(1.+rvsp)).execute(), bins=50, normed=True)
>>> plt.plot(xx.execute(),powpdf,'r-')
>>> plt.title('inverse of 1 + np.random.pareto(5)')
```

```
>>> plt.figure()
>>> plt.hist((1./(1.+rvsp)).execute(), bins=50, normed=True)
>>> plt.plot(xx.execute(),powpdf,'r-')
>>> plt.title('inverse of stats.pareto(5)')
```

mars.tensor.random.rayleigh

mars.tensor.random.**rayleigh = <bound method rayleigh of <mars.tensor.random.core.RandomStat**

Draw samples from a Rayleigh distribution.

The χ and Weibull distributions are generalizations of the Rayleigh.

scale [float or array_like of floats, optional] Scale, also equals the mode. Should be >= 0. Default is 1.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `scale` is a scalar. Otherwise, `mt.array(scale).size` samples are drawn.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized Rayleigh distribution.

The probability density function for the Rayleigh distribution is

$$P(x;scale) = \frac{x}{scale^2}e^{\frac{-x^2}{2\cdot scale^2}}$$

The Rayleigh distribution would arise, for example, if the East and North components of the wind velocity had identical zero-mean Gaussian distributions. Then the wind speed would have a Rayleigh distribution.

Draw values from the distribution and plot the histogram

```
>>> import matplotlib.pyplot as plt
>>> import mars.tensor as mt
```

```
>>> values = plt.hist(mt.random.rayleigh(3, 100000).execute(), bins=200,␣
↪normed=True)
```

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```
>>> meanvalue = 1
>>> modevalue = mt.sqrt(2 / mt.pi) * meanvalue
>>> s = mt.random.rayleigh(modevalue, 1000000)
```

The percentage of waves larger than 3 meters is:

```
>>> (100.*mt.sum(s>3)/1000000.).execute()
0.087300000000000003
```

mars.tensor.random.standard_cauchy

mars.tensor.random.**standard_cauchy = <bound method standard_cauchy of <mars.tensor.random.**

Draw samples from a standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

samples [Tensor or scalar] The drawn samples.

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi\gamma\left[1 + \left(\frac{x-x_0}{\gamma}\right)^2\right]}$$

and the Standard Cauchy distribution just sets $x_0 = 0$ and $\gamma = 1$

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

Draw samples and plot the distribution:

```
>>> import mars.tensor as mt
>>> import matplotlib.pyplot as plt
```

```
>>> s = mt.random.standard_cauchy(1000000)
>>> s = s[(s>-25) & (s<25)]  # truncate distribution so it plots well
>>> plt.hist(s.execute(), bins=100)
>>> plt.show()
```

mars.tensor.random.standard_exponential

mars.tensor.random.**standard_exponential = <bound method standard_exponential of <mars.tens**

Draw samples from the standard exponential distribution.

standard_exponential is identical to the exponential distribution with a scale parameter of 1.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [float or Tensor] Drawn samples.

Output a 3x8000 tensor:

```
>>> import mars.tensor as mt
>>> n = mt.random.standard_exponential((3, 8000))
```

mars.tensor.random.standard_gamma

mars.tensor.random.**standard_gamma = <bound method standard_gamma of <mars.tensor.random.co**

Draw samples from a standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated "k") and scale=1.

shape [float or array_like of floats] Parameter, should be > 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `shape` is a scalar. Otherwise, `mt.array(shape).size` samples are drawn.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized standard gamma distribution.

scipy.stats.gamma [probability density function, distribution or] cumulative density function, etc.

The probability density for the Gamma distribution is

$$p(x) = x^{k-1}\frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

Draw samples from the distribution:

```
>>> import mars.tensor as mt
```

```
>>> shape, scale = 2., 1. # mean and width
>>> s = mt.random.standard_gamma(shape, 1000000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s.execute(), 50, normed=True)
>>> y = bins**(shape-1) * ((mt.exp(-bins/scale))/ \
...                       (sps.gamma(shape) * scale**shape))
>>> plt.plot(bins, y.execute(), linewidth=2, color='r')
>>> plt.show()
```

mars.tensor.random.standard_normal

mars.tensor.random.**standard_normal = <bound method standard_normal of <mars.tensor.random.**

Draw samples from a standard Normal distribution (mean=0, stdev=1).

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [float or Tensor] Drawn samples.

```
>>> import mars.tensor as mt
```

```
>>> s = mt.random.standard_normal(8000)
>>> s.execute()
array([ 0.6888893 ,  0.78096262, -0.89086505, ...,  0.49876311, #random
       -0.38672696, -0.4685006 ])                               #random
>>> s.shape
(8000,)
>>> s = mt.random.standard_normal(size=(3, 4, 2))
>>> s.shape
(3, 4, 2)
```

mars.tensor.random.standard_t

mars.tensor.random.**standard_t = <bound method standard_t of <mars.tensor.random.core.Rando**

Draw samples from a standard Student's t distribution with *df* degrees of freedom.

A special case of the hyperbolic distribution. As *df* gets large, the result resembles that of the standard normal distribution (*standard_normal*).

df [float or array_like of floats] Degrees of freedom, should be > 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `df` is a scalar. Otherwise, `mt.array(df).size` samples are drawn.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized standard Student's t distribution.

The probability density function for the t distribution is

$$P(x, df) = \frac{\Gamma(\frac{df+1}{2})}{\sqrt{\pi df}\,\Gamma(\frac{df}{2})}\left(1 + \frac{x^2}{df}\right)^{-(df+1)/2}$$

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was first published in 1908 by William Gosset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

From Dalgaard page 83[1], suppose the daily energy intake for 11 women in Kj is:

```
>>> import mars.tensor as mt
```

```
>>> intake = mt.array([5260., 5470, 5640, 6180, 6390, 6515, 6805, 7515, \
...                    7515, 8230, 8770])
```

Does their energy intake deviate systematically from the recommended value of 7725 kJ?

We have 10 degrees of freedom, so is the sample mean within 95% of the recommended value?

```
>>> s = mt.random.standard_t(10, size=100000)
>>> mt.mean(intake).execute()
6753.636363636364
>>> intake.std(ddof=1).execute()
1142.1232221373727
```

Calculate the t statistic, setting the ddof parameter to the unbiased value so the divisor in the standard deviation will be degrees of freedom, N-1.

```
>>> t = (mt.mean(intake)-7725)/(intake.std(ddof=1)/mt.sqrt(len(intake)))
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(s.execute(), bins=100, normed=True)
```

For a one-sided t-test, how far out in the distribution does the t statistic appear?

```
>>> (mt.sum(s<t) / float(len(s))).execute()
0.0090699999999999999  #random
```

So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.

[1] Dalgaard, Peter, "Introductory Statistics With R", Springer, 2002.

mars.tensor.random.triangular

mars.tensor.random.**triangular = <bound method triangular of <mars.tensor.random.core.Randor**

Draw samples from the triangular distribution over the interval `[left, right]`.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

left [float or array_like of floats] Lower limit.

mode [float or array_like of floats] The value where the peak of the distribution occurs. The value should fulfill the condition `left <= mode <= right`.

right [float or array_like of floats] Upper limit, should be larger than *left*.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `left`, `mode`, and `right` are all scalars. Otherwise, `mt.broadcast(left, mode, right).size` samples are drawn.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized triangular distribution.

The probability density function for the triangular distribution is

$$P(x;l,m,r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\ \frac{2(r-x)}{(r-l)(r-m)} & \text{for } m \leq x \leq r, \\ 0 & \text{otherwise.} \end{cases}$$

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> import mars.tensor as mt
>>> h = plt.hist(mt.random.triangular(-3, 0, 8, 100000).execute(), bins=200,
...              normed=True)
>>> plt.show()
```

mars.tensor.random.uniform

mars.tensor.random.**uniform = <bound method uniform of <mars.tensor.random.core.RandomState**

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval `[low, high)` (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

low [float or array_like of floats, optional] Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

high [float or array_like of floats] Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `low` and `high` are both scalars. Otherwise, `mt.broadcast(low, high).size` samples are drawn.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized uniform distribution.

randint : Discrete uniform distribution, yielding integers. random_integers : Discrete uniform distribution over the closed

interval `[low, high]`.

random_sample : Floats uniformly distributed over `[0, 1)`. random : Alias for *random_sample*. rand : Convenience function that accepts dimensions as input, e.g.,

`rand(2,2)` would generate a 2-by-2 array of floats, uniformly distributed over `[0, 1)`.

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b - a}$$

anywhere within the interval `[a, b)`, and zero elsewhere.

When `high` == `low`, values of `low` will be returned. If `high` < `low`, the results are officially undefined and may eventually raise an error, i.e. do not rely on this function to behave when passed arguments satisfying that inequality condition.

Draw samples from the distribution:

```
>>> import mars.tensor as mt
```

```
>>> s = mt.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> mt.all(s >= -1).execute()
True
>>> mt.all(s < 0).execute()
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s.execute(), 15, normed=True)
>>> plt.plot(bins, mt.ones_like(bins).execute(), linewidth=2, color='r')
>>> plt.show()
```

mars.tensor.random.vonmises

mars.tensor.random.**vonmises = <bound method vonmises of <mars.tensor.random.core.RandomSta**

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

mu [float or array_like of floats] Mode ("center") of the distribution.

kappa [float or array_like of floats] Dispersion of the distribution, has to be >=0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `mu` and `kappa` are both scalars. Otherwise, `np.broadcast(mu, kappa).size` samples are drawn.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized von Mises distribution.

scipy.stats.vonmises [probability density function, distribution, or] cumulative density function, etc.

The probability density for the von Mises distribution is

$$p(x) = \frac{e^{\kappa cos(x-\mu)}}{2\pi I_0(\kappa)},$$

where μ is the mode and κ the dispersion, and $I_0(\kappa)$ is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

Draw samples from the distribution:

```
>>> import mars.tensor as mt
```

```
>>> mu, kappa = 0.0, 4.0 # mean and dispersion
>>> s = mt.random.vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> from scipy.special import i0
>>> plt.hist(s.execute(), 50, normed=True)
>>> x = mt.linspace(-mt.pi, mt.pi, num=51)
>>> y = mt.exp(kappa*mt.cos(x-mu))/(2*mt.pi*i0(kappa))
>>> plt.plot(x.execute(), y.execute(), linewidth=2, color='r')
>>> plt.show()
```

mars.tensor.random.wald

mars.tensor.random.**wald = <bound method wald of <mars.tensor.random.core.RandomState object**

Draw samples from a Wald, or inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian. Some references claim that the Wald is an inverse Gaussian with mean equal to 1, but this is by no means universal.

The inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

mean [float or array_like of floats] Distribution mean, should be > 0.

scale [float or array_like of floats] Scale parameter, should be >= 0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `mean` and `scale` are both scalars. Otherwise, `np.broadcast(mean, scale).size` samples are drawn.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized Wald distribution.

The probability density function for the Wald distribution is

$$P(x; mean, scale) = \sqrt{\frac{scale}{2\pi x^3}} e^{\frac{-scale(x-mean)^2}{2 \cdot mean^2 x}}$$

As noted above the inverse Gaussian distribution first arise from attempts to model Brownian motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> import mars.tensor as mt
>>> h = plt.hist(mt.random.wald(3, 2, 100000).execute(), bins=200, normed=True)
>>> plt.show()
```

mars.tensor.random.weibull

mars.tensor.random.**weibull = <bound method weibull of <mars.tensor.random.core.RandomState**

Draw samples from a Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter a.

$$X = (-ln(U))^{1/a}$$

Here, U is drawn from the uniform distribution over (0,1].

The more common 2-parameter Weibull, including a scale parameter λ is just $X = \lambda(-ln(U))^{1/a}$.

a [float or array_like of floats] Shape of the distribution. Should be greater than zero.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `mt.array(a).size` samples are drawn.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized Weibull distribution.

scipy.stats.weibull_max scipy.stats.weibull_min scipy.stats.genextreme gumbel

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

$$p(x) = \frac{a}{\lambda}(\frac{x}{\lambda})^{a-1}e^{-(x/\lambda)^a},$$

where a is the shape and λ the scale.

The function has its peak (the mode) at $\lambda(\frac{a-1}{a})^{1/a}$.

When `a = 1`, the Weibull distribution reduces to the exponential distribution.

Draw samples from the distribution:

```
>>> import mars.tensor as mt
```

```
>>> a = 5. # shape
>>> s = mt.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> x = mt.arange(1,100.)/50.
>>> def weib(x,n,a):
...     return (a / n) * (x / n)**(a - 1) * mt.exp(-(x / n)**a)
```

```
>>> count, bins, ignored = plt.hist(mt.random.weibull(5.,1000).execute())
>>> x = mt.arange(1,100.)/50.
>>> scale = count.max()/weib(x, 1., 5.).max()
>>> plt.plot(x.execute(), (weib(x, 1., 5.)*scale).execute())
>>> plt.show()
```

mars.tensor.random.zipf

mars.tensor.random.**zipf = <bound method zipf of <mars.tensor.random.core.RandomState object**

Draw samples from a Zipf distribution.

Samples are drawn from a Zipf distribution with specified parameter $a > 1$.

The Zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf's law: the frequency of an item is inversely proportional to its rank in a frequency table.

a [float or array_like of floats] Distribution parameter. Should be greater than 1.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `mt.array(a).size` samples are drawn.

chunk_size [int or tuple of int or tuple of ints, optional] Desired chunk size on each dimension

gpu [bool, optional] Allocate the tensor on GPU if True, False as default

dtype [data-type, optional] Data-type of the returned tensor.

out [Tensor or scalar] Drawn samples from the parameterized Zipf distribution.

scipy.stats.zipf [probability density function, distribution, or] cumulative density function, etc.

The probability density for the Zipf distribution is

$$p(x) = \frac{x^{-a}}{\zeta(a)},$$

where ζ is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

Draw samples from the distribution:

```
>>> import mars.tensor as mt
```

```
>>> a = 2. # parameter
>>> s = mt.random.zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> from scipy import special
```

Truncate s values at 50 so plot is interesting:

```
>>> count, bins, ignored = plt.hist(s[s<50].execute(), 50, normed=True)
>>> x = mt.arange(1., 50.)
>>> y = x**(-a) / special.zetac(a)
>>> plt.plot(x.execute(), (y/mt.max(y)).execute(), linewidth=2, color='r')
>>> plt.show()
```

Random number generator

mars.tensor.random.seed	Seed the generator.
mars.tensor.random.RandomState	

mars.tensor.random.seed

mars.tensor.random.**seed = <bound method RandomState.seed of <mars.tensor.random.core.Randor**

Seed the generator.

This method is called when *RandomState* is initialized. It can be called again to re-seed the generator. For details, see *RandomState*.

seed [int or 1-d array_like, optional] Seed for *RandomState*. Must be convertible to 32 bit unsigned integers.

RandomState

mars.tensor.random.RandomState

class mars.tensor.random.**RandomState**(*seed=None*)

__init__(*seed=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([seed])	Initialize self.
beta(a, b[, size, chunk_size, gpu, dtype])	Draw samples from a Beta distribution.
binomial(n, p[, size, chunk_size, gpu, dtype])	Draw samples from a binomial distribution.
bytes(length)	Return random bytes.
chisquare(df[, size, chunk_size, gpu, dtype])	Draw samples from a chi-square distribution.
choice(a[, size, replace, p, chunk_size, gpu])	Generates a random sample from a given 1-D array
dirichlet(alpha[, size, chunk_size, gpu, dtype])	Draw samples from the Dirichlet distribution.
exponential([scale, size, chunk_size, gpu, ...])	Draw samples from an exponential distribution.
f(dfnum, dfden[, size, chunk_size, gpu, dtype])	Draw samples from an F distribution.
gamma(shape[, scale, size, chunk_size, gpu, ...])	Draw samples from a Gamma distribution.
geometric(p[, size, chunk_size, gpu, dtype])	Draw samples from the geometric distribution.
gumbel([loc, scale, size, chunk_size, gpu, ...])	Draw samples from a Gumbel distribution.
hypergeometric(ngood, nbad, nsample[, size, ...])	Draw samples from a Hypergeometric distribution.
laplace([loc, scale, size, chunk_size, gpu, ...])	Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).
logistic([loc, scale, size, chunk_size, ...])	Draw samples from a logistic distribution.
lognormal([mean, sigma, size, chunk_size, ...])	Draw samples from a log-normal distribution.
logseries(p[, size, chunk_size, gpu, dtype])	Draw samples from a logarithmic series distribution.
multinomial(n, pvals[, size, chunk_size, ...])	Draw samples from a multinomial distribution.
multivariate_normal(mean, cov[, size, ...])	Draw random samples from a multivariate normal distribution.
negative_binomial(n, p[, size, chunk_size, ...])	Draw samples from a negative binomial distribution.
noncentral_chisquare(df, nonc[, size, ...])	Draw samples from a noncentral chi-square distribution.
noncentral_f(dfnum, dfden, nonc[, size, ...])	Draw samples from the noncentral F distribution.
normal([loc, scale, size, chunk_size, gpu, ...])	Draw random samples from a normal (Gaussian) distribution.
pareto(a[, size, chunk_size, gpu, dtype])	Draw samples from a Pareto II or Lomax distribution with specified shape.
poisson([lam, size, chunk_size, gpu, dtype])	Draw samples from a Poisson distribution.
power(a[, size, chunk_size, gpu, dtype])	Draws samples in [0, 1] from a power distribution with positive exponent a - 1.
rand(*dn, **kw)	Random values in a given shape.
randint(low[, high, size, dtype, density, ...])	Return random integers from *low* (inclusive) to *high* (exclusive).
randn(*dn, **kw)	Return a sample (or samples) from the "standard normal" distribution.
random([size, chunk_size, gpu, dtype])	Return random floats in the half-open interval [0.0, 1.0).
random_integers(low[, high, size, ...])	Random integers of type mt.int between *low* and *high*, inclusive.

Continued on next page

Table 48 – continued from previous page

random_sample([size, chunk_size, gpu, dtype])	Return random floats in the half-open interval [0.0, 1.0).
ranf([size, chunk_size, gpu, dtype])	Return random floats in the half-open interval [0.0, 1.0).
rayleigh([scale, size, chunk_size, gpu, dtype])	Draw samples from a Rayleigh distribution.
sample([size, chunk_size, gpu, dtype])	Return random floats in the half-open interval [0.0, 1.0).
seed([seed])	Seed the generator.
standard_cauchy([size, chunk_size, gpu, dtype])	Draw samples from a standard Cauchy distribution with mode = 0.
standard_exponential([size, chunk_size, ...])	Draw samples from the standard exponential distribution.
standard_gamma(shape[, size, chunk_size, ...])	Draw samples from a standard Gamma distribution.
standard_normal([size, chunk_size, gpu, dtype])	Draw samples from a standard Normal distribution (mean=0, stdev=1).
standard_t(df[, size, chunk_size, gpu, dtype])	Draw samples from a standard Student's t distribution with *df* degrees of freedom.
triangular(left, mode, right[, size, ...])	Draw samples from the triangular distribution over the interval `[left, right]`.
uniform([low, high, size, chunk_size, gpu, ...])	Draw samples from a uniform distribution.
vonmises(mu, kappa[, size, chunk_size, gpu, ...])	Draw samples from a von Mises distribution.
wald(mean, scale[, size, chunk_size, gpu, dtype])	Draw samples from a Wald, or inverse Gaussian, distribution.
weibull(a[, size, chunk_size, gpu, dtype])	Draw samples from a Weibull distribution.
zipf(a[, size, chunk_size, gpu, dtype])	Draw samples from a Zipf distribution.

2.6.10 Set routines

Boolean operations

mars.tensor.isin	Calculates *element in test_elements*, broadcasting over *element* only.

mars.tensor.isin

mars.tensor.**isin**(*element*, *test_elements*, *assume_unique=False*, *invert=False*)

Calculates *element in test_elements*, broadcasting over *element* only. Returns a boolean array of the same shape as *element* that is True where an element of *element* is in *test_elements* and False otherwise.

element [array_like] Input tensor.

test_elements [array_like] The values against which to test each value of *element*. This argument is flattened if it is a tensor or array_like. See notes for behavior with non-array-like parameters.

assume_unique [bool, optional] If True, the input tensors are both assumed to be unique, which can speed up the calculation. Default is False.

invert [bool, optional] If True, the values in the returned tensor are inverted, as if calculating *element not in test_elements*. Default is False. `mt.isin(a, b, invert=True)` is equivalent to (but faster than) `mt.invert(mt.isin(a, b))`.

isin [Tensor, bool] Has the same shape as *element*. The values *element[isin]* are in *test_elements*.

in1d : Flattened version of this function.

isin is an element-wise function version of the python keyword *in*. `isin(a, b)` is roughly equivalent to `mt.array([item in b for item in a])` if *a* and *b* are 1-D sequences.

element and *test_elements* are converted to tensors if they are not already. If *test_elements* is a set (or other non-sequence collection) it will be converted to an object tensor with one element, rather than a tensor of the values contained in *test_elements*. This is a consequence of the *tensor* constructor's way of handling non-sequence collections. Converting the set to a list usually gives the desired behavior.

```
>>> import mars.tensor as mt
```

```
>>> element = 2*mt.arange(4).reshape((2, 2))
>>> element.execute()
array([[0, 2],
       [4, 6]])
>>> test_elements = [1, 2, 4, 8]
>>> mask = mt.isin(element, test_elements)
>>> mask.execute()
array([[ False,  True],
       [ True,  False]])
>>> element[mask].execute()
array([2, 4])
>>> mask = mt.isin(element, test_elements, invert=True)
>>> mask.execute()
array([[ True, False],
       [ False, True]])
>>> element[mask]
array([0, 6])
```

Because of how *array* handles sets, the following does not work as expected:

```
>>> test_set = {1, 2, 4, 8}
>>> mt.isin(element, test_set).execute()
array([[ False, False],
       [ False, False]])
```

Casting the set to a list gives the expected result:

```
>>> mt.isin(element, list(test_set)).execute()
array([[ False,  True],
       [ True,  False]])
```

2.6.11 Sorting, Searching, and Counting

Searching

mars.tensor.argmax	Returns the indices of the maximum values along an axis.
mars.tensor.nanargmax	Return the indices of the maximum values in the specified axis ignoring NaNs.
mars.tensor.argmin	Returns the indices of the minimum values along an axis.

Continued on next page

Table 50 – continued from previous page

mars.tensor.nanargmin	Return the indices of the minimum values in the specified axis ignoring NaNs.
mars.tensor.argwhere	Find the indices of tensor elements that are non-zero, grouped by element.
mars.tensor.argmin	Returns the indices of the minimum values along an axis.
mars.tensor.nonzero	Return the indices of the elements that are non-zero.
mars.tensor.flatnonzero	Return indices that are non-zero in the flattened version of a.
mars.tensor.where	Return elements, either from *x* or *y*, depending on *condition*.
mars.tensor.extract	Return the elements of a tensor that satisfy some condition.

mars.tensor.argmax

mars.tensor.**argmax**(*a*, *axis=None*, *out=None*, *combine_size=None*)

Returns the indices of the maximum values along an axis.

a [array_like] Input tensor.

axis [int, optional] By default, the index is into the flattened tensor, otherwise along the specified axis.

out [Tensor, optional] If provided, the result will be inserted into this tensor. It should be of the appropriate shape and dtype.

combine_size: int, optional The number of chunks to combine.

index_array [Tensor of ints] Tensor of indices into the tensor. It has the same shape as *a.shape* with the dimension along *axis* removed.

Tensor.argmax, argmin amax : The maximum value along a given axis. unravel_index : Convert a flat index into an index tuple.

In case of multiple occurrences of the maximum values, the indices corresponding to the first occurrence are returned.

```
>>> import mars.tensor as mt
>>> from mars.session import new_session
```

```
>>> sess = new_session().as_default()
```

```
>>> a = mt.arange(6).reshape(2,3)
>>> a.execute()
array([[0, 1, 2],
       [3, 4, 5]])
>>> mt.argmax(a).execute()
5
>>> mt.argmax(a, axis=0).execute()
array([1, 1, 1])
>>> mt.argmax(a, axis=1).execute()
array([2, 2])
```

Indexes of the maximal elements of a N-dimensional tensor:

```
>>> ind = mt.unravel_index(mt.argmax(a, axis=None), a.shape)
>>> sess.run(ind)
(1, 2)
>>> a[ind].execute()  # TODO(jisheng): accomplish when fancy index on tensor is␣
↪supported
```

```
>>> b = mt.arange(6)
>>> b[1] = 5
>>> b.execute()
array([0, 5, 2, 3, 4, 5])
>>> mt.argmax(b).execute()  # Only the first occurrence is returned.
1
```

mars.tensor.nanargmax

mars.tensor.**nanargmax**(*a*, *axis=None*, *out=None*, *combine_size=None*)

Return the indices of the maximum values in the specified axis ignoring NaNs. For all-NaN slices `ValueError` is raised. Warning: the results cannot be trusted if a slice contains only NaNs and -Infs.

a [array_like] Input data.

axis [int, optional] Axis along which to operate. By default flattened input is used.

out [Tensor, optional] Alternate output tensor in which to place the result. The default is `None`; if provided, it must have the same shape as the expected output, but the type will be cast if necessary. See *doc.ufuncs* for details.

combine_size: int, optional The number of chunks to combine.

index_array [Tensor] An tensor of indices or a single index value.

argmax, nanargmin

```
>>> import mars.tensor as mt
```

```
>>> a = mt.array([[mt.nan, 4], [2, 3]])
>>> mt.argmax(a).execute()
0
>>> mt.nanargmax(a).execute()
1
>>> mt.nanargmax(a, axis=0).execute()
array([1, 0])
>>> mt.nanargmax(a, axis=1).execute()
array([1, 1])
```

mars.tensor.argmin

mars.tensor.**argmin**(*a*, *axis=None*, *out=None*, *combine_size=None*)

Returns the indices of the minimum values along an axis.

a [array_like] Input tensor.

axis [int, optional] By default, the index is into the flattened tensor, otherwise along the specified axis.

out [Tensor, optional] If provided, the result will be inserted into this tensor. It should be of the appropriate shape and dtype.

combine_size: int, optional The number of chunks to combine.

index_array [Tensor of ints] Tensor of indices into the tensor. It has the same shape as *a.shape* with the dimension along *axis* removed.

Tensor.argmin, argmax amin : The minimum value along a given axis. unravel_index : Convert a flat index into an index tuple.

In case of multiple occurrences of the minimum values, the indices corresponding to the first occurrence are returned.

```
>>> import mars.tensor as mt
>>> from mars.session import new_session
```

```
>>> sess = new_session().as_default()
```

```
>>> a = mt.arange(6).reshape(2,3)
>>> a.execute()
array([[0, 1, 2],
       [3, 4, 5]])
>>> mt.argmin(a).execute()
0
>>> mt.argmin(a, axis=0).execute()
array([0, 0, 0])
>>> mt.argmin(a, axis=1).execute()
array([0, 0])
```

Indices of the minimum elements of a N-dimensional tensor:

```
>>> ind = mt.unravel_index(mt.argmin(a, axis=None), a.shape)
>>> sess.run(ind)
(0, 0)
>>> a[ind]  # TODO(jisheng): accomplish when fancy index on tensor is supported
```

```
>>> b = mt.arange(6)
>>> b[4] = 0
>>> b.execute()
array([0, 1, 2, 3, 0, 5])
>>> mt.argmin(b).execute()  # Only the first occurrence is returned.
0
```

mars.tensor.nanargmin

mars.tensor.**nanargmin**(*a*, *axis=None*, *out=None*, *combine_size=None*)

Return the indices of the minimum values in the specified axis ignoring NaNs. For all-NaN slices `ValueError` is raised. Warning: the results cannot be trusted if a slice contains only NaNs and Infs.

a [array_like] Input data.

axis [int, optional] Axis along which to operate. By default flattened input is used.

combine_size: int, optional The number of chunks to combine.

index_array [Tensor] A tensor of indices or a single index value.

argmin, nanargmax

```
>>> import mars.tensor as mt
```

```
>>> a = mt.array([[mt.nan, 4], [2, 3]])
>>> mt.argmin(a).execute()
0
>>> mt.nanargmin(a).execute()
2
>>> mt.nanargmin(a, axis=0).execute()
array([1, 1])
>>> mt.nanargmin(a, axis=1).execute()
array([1, 0])
```

mars.tensor.argwhere

`mars.tensor.`**`argwhere`**`(a)`

Find the indices of tensor elements that are non-zero, grouped by element.

a [array_like] Input data.

index_tensor [Tensor] Indices of elements that are non-zero. Indices are grouped by element.

where, nonzero

`mt.argwhere(a)` is the same as `mt.transpose(mt.nonzero(a))`.

The output of `argwhere` is not suitable for indexing tensors. For this purpose use `nonzero(a)` instead.

```
>>> import mars.tensor as mt
```

```
>>> x = mt.arange(6).reshape(2,3)
>>> x.execute()
array([[0, 1, 2],
       [3, 4, 5]])
>>> mt.argwhere(x>1).execute()
array([[0, 2],
       [1, 0],
       [1, 1],
       [1, 2]])
```

mars.tensor.flatnonzero

`mars.tensor.`**`flatnonzero`**`(a)`

Return indices that are non-zero in the flattened version of a.

This is equivalent to a.ravel().nonzero()[0].

a [Tensor] Input tensor.

res [Tensor] Output tensor, containing the indices of the elements of *a.ravel()* that are non-zero.

nonzero : Return the indices of the non-zero elements of the input tensor. ravel : Return a 1-D tensor containing the elements of the input tensor.

```
>>> import mars.tensor as mt
```

```
>>> x = mt.arange(-2, 3)
>>> x.execute()
array([-2, -1,  0,  1,  2])
>>> mt.flatnonzero(x).execute()
array([0, 1, 3, 4])
```

Use the indices of the non-zero elements as an index array to extract these elements:

```
>>> x.ravel()[mt.flatnonzero(x)].execute()  # TODO(jisheng): accomplish this␣
↪after fancy indexing is supported
```

mars.tensor.extract

mars.tensor.**extract**(*condition*, *a*)

Return the elements of a tensor that satisfy some condition.

This is equivalent to `mt.compress(ravel(condition), ravel(arr))`. If *condition* is boolean `mt.extract` is equivalent to `arr[condition]`.

Note that *place* does the exact opposite of *extract*.

condition [array_like] An array whose nonzero or True entries indicate the elements of *arr* to extract.

a [array_like] Input tensor of the same size as *condition*.

extract [Tensor] Rank 1 tensor of values from *arr* where *condition* is True.

take, put, copyto, compress, place

```
>>> import mars.tensor as mt
```

```
>>> arr = mt.arange(12).reshape((3, 4))
>>> arr.execute()
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> condition = mt.mod(arr, 3)==0
>>> condition.execute()
array([[ True, False, False,  True],
       [False, False,  True, False],
       [False,  True, False, False]])
>>> mt.extract(condition, arr).execute()
array([0, 3, 6, 9])
```

If *condition* is boolean:

```
>>> arr[condition].execute()
array([0, 3, 6, 9])
```

Counting

mars.tensor.count_nonzero	Counts the number of non-zero values in the tensor `a`.

mars.tensor.count_nonzero

`mars.tensor.`**`count_nonzero`**(*a*, *axis=None*, *combine_size=None*)

Counts the number of non-zero values in the tensor `a`.

The word "non-zero" is in reference to the Python 2.x built-in method `__nonzero__()` (renamed `__bool__()` in Python 3.x) of Python objects that tests an object's "truthfulness". For example, any number is considered truthful if it is nonzero, whereas any string is considered truthful if it is not the empty string. Thus, this function (recursively) counts how many elements in `a` (and in sub-tensors thereof) have their `__nonzero__()` or `__bool__()` method evaluated to `True`.

a [array_like] The tensor for which to count non-zeros.

axis [int or tuple, optional] Axis or tuple of axes along which to count non-zeros. Default is None, meaning that non-zeros will be counted along a flattened version of `a`.

combine_size: int, optional The number of chunks to combine.

count [int or tensor of int] Number of non-zero values in the array along a given axis. Otherwise, the total number of non-zero values in the tensor is returned.

nonzero : Return the coordinates of all the non-zero values.

```
>>> import mars.tensor as mt
```

```
>>> mt.count_nonzero(mt.eye(4)).execute()
4
>>> mt.count_nonzero([[0,1,7,0,0],[3,0,0,2,19]]).execute()
5
>>> mt.count_nonzero([[0,1,7,0,0],[3,0,0,2,19]], axis=0).execute()
array([1, 1, 1, 1, 1])
>>> mt.count_nonzero([[0,1,7,0,0],[3,0,0,2,19]], axis=1).execute()
array([2, 3])
```

2.6.12 Statistics

Order statistics

mars.tensor.amin	Return the minimum of a tensor or minimum along an axis.
mars.tensor.amax	Return the maximum of an array or maximum along an axis.
mars.tensor.nanmin	Return minimum of a tensor or minimum along an axis, ignoring any NaNs.
mars.tensor.nanmax	Return the maximum of an array or maximum along an axis, ignoring any NaNs.
mars.tensor.ptp	Range of values (maximum - minimum) along an axis.

mars.tensor.amin

mars.tensor.**amin**(*a*, *axis=None*, *out=None*, *keepdims=None*, *combine_size=None*)

Return the minimum of a tensor or minimum along an axis.

a [array_like] Input data.

axis [None or int or tuple of ints, optional] Axis or axes along which to operate. By default, flattened input is used.

If this is a tuple of ints, the minimum is selected over multiple axes, instead of a single axis or all the axes as before.

out [Tensor, optional] Alternative output tensor in which to place the result. Must be of the same shape and buffer length as the expected output. See *doc.ufuncs* (Section "Output arguments") for more details.

keepdims [bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input tensor.

If the default value is passed, then *keepdims* will not be passed through to the *amin* method of sub-classes of *Tensor*, however any non-default value will be. If the sub-classes *sum* method does not implement *keepdims* any exceptions will be raised.

combine_size: int, optional The number of chunks to combine.

amin [Tensor or scalar] Minimum of *a*. If *axis* is None, the result is a scalar value. If *axis* is given, the result is an array of dimension `a.ndim - 1`.

amax : The maximum value of a tensor along a given axis, propagating any NaNs.

nanmin : The minimum value of a tensor along a given axis, ignoring any NaNs.

minimum : Element-wise minimum of two tensors, propagating any NaNs.

fmin : Element-wise minimum of two tensors, ignoring any NaNs.

argmin : Return the indices of the minimum values.

nanmax, maximum, fmax

NaN values are propagated, that is if at least one item is NaN, the corresponding min value will be NaN as well. To ignore NaN values (MATLAB behavior), please use nanmin.

Don't use *amin* for element-wise comparison of 2 tensors; when `a.shape[0]` is 2, `minimum(a[0], a[1])` is faster than `amin(a, axis=0)`.

```
>>> import mars.tensor as mt
```

```
>>> a = mt.arange(4).reshape((2,2))
>>> a.execute()
array([[0, 1],
       [2, 3]])
>>> mt.amin(a).execute()           # Minimum of the flattened array
0
>>> mt.amin(a, axis=0).execute()   # Minima along the first axis
array([0, 1])
>>> mt.amin(a, axis=1).execute()   # Minima along the second axis
array([0, 2])
```

```
>>> b = mt.arange(5, dtype=float)
>>> b[2] = mt.NaN
>>> mt.amin(b).execute()
nan
>>> mt.nanmin(b).execute()
0.0
```

mars.tensor.amax

mars.tensor.**amax**(*a*, *axis=None*, *out=None*, *keepdims=None*, *combine_size=None*)

Return the maximum of an array or maximum along an axis.

a [array_like] Input data.

axis [None or int or tuple of ints, optional] Axis or axes along which to operate. By default, flattened input is used.

If this is a tuple of ints, the maximum is selected over multiple axes, instead of a single axis or all the axes as before.

out [Tensor, optional] Alternative output tensor in which to place the result. Must be of the same shape and buffer length as the expected output. See *doc.ufuncs* (Section "Output arguments") for more details.

keepdims [bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

If the default value is passed, then *keepdims* will not be passed through to the *amax* method of sub-classes of *ndarray*, however any non-default value will be. If the sub-classes *sum* method does not implement *keepdims* any exceptions will be raised.

combine_size: int, optional The number of chunks to combine.

amax [Tensor or scalar] Maximum of *a*. If *axis* is None, the result is a scalar value. If *axis* is given, the result is a tensor of dimension `a.ndim - 1`.

amin : The minimum value of a tensor along a given axis, propagating any NaNs.

nanmax : The maximum value of a tensor along a given axis, ignoring any NaNs.

maximum : Element-wise maximum of two tensors, propagating any NaNs.

fmax : Element-wise maximum of two tensors, ignoring any NaNs.

argmax : Return the indices of the maximum values.

nanmin, minimum, fmin

NaN values are propagated, that is if at least one item is NaN, the corresponding max value will be NaN as well. To ignore NaN values (MATLAB behavior), please use nanmax.

Don't use *amax* for element-wise comparison of 2 arrays; when `a.shape[0]` is 2, `maximum(a[0], a[1])` is faster than `amax(a, axis=0)`.

```
>>> import mars.tensor as mt
```

```
>>> a = mt.arange(4).reshape((2,2))
>>> a.execute()
array([[0, 1],
```

(continues on next page)

(continued from previous page)

```
       [2, 3]])
>>> mt.amax(a).execute()           # Maximum of the flattened array
3
>>> mt.amax(a, axis=0).execute()   # Maxima along the first axis
array([2, 3])
>>> mt.amax(a, axis=1).execute()   # Maxima along the second axis
array([1, 3])
```

```
>>> b = mt.arange(5, dtype=float)
>>> b[2] = mt.NaN
>>> mt.amax(b).execute()
nan
>>> mt.nanmax(b).execute()
4.0
```

mars.tensor.nanmin

mars.tensor.**nanmin**(*a*, *axis=None*, *out=None*, *keepdims=None*, *combine_size=None*)

Return minimum of a tensor or minimum along an axis, ignoring any NaNs. When all-NaN slices are encountered a `RuntimeWarning` is raised and Nan is returned for that slice.

a [array_like] Tensor containing numbers whose minimum is desired. If *a* is not an tensor, a conversion is attempted.

axis [int, optional] Axis along which the minimum is computed. The default is to compute the minimum of the flattened tensor.

out [Tensor, optional] Alternate output tensor in which to place the result. The default is `None`; if provided, it must have the same shape as the expected output, but the type will be cast if necessary. See *doc.ufuncs* for details.

keepdims [bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *a*.

If the value is anything but the default, then *keepdims* will be passed through to the *min* method of sub-classes of *Tensor*. If the sub-classes methods does not implement *keepdims* any exceptions will be raised.

combine_size: int, optional The number of chunks to combine.

nanmin [Tensor] An tensor with the same shape as *a*, with the specified axis removed. If *a* is a 0-d tensor, or if axis is None, a tensor scalar is returned. The same dtype as *a* is returned.

nanmax : The maximum value of an array along a given axis, ignoring any NaNs.

amin : The minimum value of an array along a given axis, propagating any NaNs.

fmin : Element-wise minimum of two arrays, ignoring any NaNs.

minimum : Element-wise minimum of two arrays, propagating any NaNs.

isnan : Shows which elements are Not a Number (NaN).

isfinite: Shows which elements are neither NaN nor infinity.

amax, fmax, maximum

Mars uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity. Positive infinity is treated as a very large number and negative infinity is treated as a very small (i.e. negative) number.

If the input has a integer type the function is equivalent to mt.min.

```
>>> import mars.tensor as mt
```

```
>>> a = mt.array([[1, 2], [3, mt.nan]])
>>> mt.nanmin(a).execute()
1.0
>>> mt.nanmin(a, axis=0).execute()
array([ 1.,  2.])
>>> mt.nanmin(a, axis=1).execute()
array([ 1.,  3.])
```

When positive infinity and negative infinity are present:

```
>>> mt.nanmin([1, 2, mt.nan, mt.inf]).execute()
1.0
>>> mt.nanmin([1, 2, mt.nan, mt.NINF]).execute()
-inf
```

mars.tensor.nanmax

mars.tensor.**nanmax**(*a*, *axis=None*, *out=None*, *keepdims=None*, *combine_size=None*)

Return the maximum of an array or maximum along an axis, ignoring any NaNs. When all-NaN slices are encountered a `RuntimeWarning` is raised and NaN is returned for that slice.

a [array_like] Tensor containing numbers whose maximum is desired. If *a* is not a tensor, a conversion is attempted.

axis [int, optional] Axis along which the maximum is computed. The default is to compute the maximum of the flattened tensor.

out [ndarray, optional] Alternate output array in which to place the result. The default is `None`; if provided, it must have the same shape as the expected output, but the type will be cast if necessary. See *doc.ufuncs* for details.

keepdims [bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *a*.

If the value is anything but the default, then *keepdims* will be passed through to the *max* method of sub-classes of *Tensor*. If the sub-classes methods does not implement *keepdims* any exceptions will be raised.

combine_size: int, optional The number of chunks to combine.

nanmax [Tensor] A tensor with the same shape as *a*, with the specified axis removed. If *a* is a 0-d tensor, or if axis is None, a Tensor scalar is returned. The same dtype as *a* is returned.

nanmin : The minimum value of a tensor along a given axis, ignoring any NaNs.

amax : The maximum value of a tensor along a given axis, propagating any NaNs.

fmax : Element-wise maximum of two tensors, ignoring any NaNs.

maximum : Element-wise maximum of two tensors, propagating any NaNs.

isnan : Shows which elements are Not a Number (NaN).

isfinite: Shows which elements are neither NaN nor infinity.

amin, fmin, minimum

Mars uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity. Positive infinity is treated as a very large number and negative infinity is treated as a very small (i.e. negative) number.

If the input has a integer type the function is equivalent to np.max.

```
>>> import mars.tensor as mt
```

```
>>> a = mt.array([[1, 2], [3, mt.nan]])
>>> mt.nanmax(a).execute()
3.0
>>> mt.nanmax(a, axis=0).execute()
array([ 3.,  2.])
>>> mt.nanmax(a, axis=1).execute()
array([ 2.,  3.])
```

When positive infinity and negative infinity are present:

```
>>> mt.nanmax([1, 2, mt.nan, mt.NINF]).execute()
2.0
>>> mt.nanmax([1, 2, mt.nan, mt.inf]).execute()
inf
```

mars.tensor.ptp

mars.tensor.**ptp**(*a*, *axis=None*, *out=None*)

Range of values (maximum - minimum) along an axis.

The name of the function comes from the acronym for 'peak to peak'.

a [array_like] Input values.

axis [int, optional] Axis along which to find the peaks. By default, flatten the array.

out [array_like] Alternative output tensor in which to place the result. It must have the same shape and buffer length as the expected output, but the type of the output values will be cast if necessary.

ptp [Tensor] A new tensor holding the result, unless *out* was specified, in which case a reference to *out* is returned.

```
>>> import mars.tensor as mt
```

```
>>> x = mt.arange(4).reshape((2,2))
>>> x.execute()
array([[0, 1],
       [2, 3]])
```

```
>>> mt.ptp(x, axis=0).execute()
array([2, 2])
```

```
>>> mt.ptp(x, axis=1).execute()
array([1, 1])
```

Average and variances

mars.tensor.average	Compute the weighted average along the specified axis.
mars.tensor.mean	Compute the arithmetic mean along the specified axis.
mars.tensor.std	Compute the standard deviation along the specified axis.
mars.tensor.var	Compute the variance along the specified axis.
mars.tensor.nanmean	Compute the arithmetic mean along the specified axis, ignoring NaNs.
mars.tensor.nanstd	Compute the standard deviation along the specified axis, while ignoring NaNs.
mars.tensor.nanvar	Compute the variance along the specified axis, while ignoring NaNs.

mars.tensor.average

mars.tensor.**average**(*a*, *axis=None*, *weights=None*, *returned=False*)

Compute the weighted average along the specified axis.

a [array_like] Tensor containing data to be averaged. If *a* is not a tensor, a conversion is attempted.

axis [None or int or tuple of ints, optional] Axis or axes along which to average *a*. The default, axis=None, will average over all of the elements of the input tensor. If axis is negative it counts from the last to the first axis.

If axis is a tuple of ints, averaging is performed on all of the axes specified in the tuple instead of a single axis or all the axes as before.

weights [array_like, optional] A tensor of weights associated with the values in *a*. Each value in *a* contributes to the average according to its associated weight. The weights tensor can either be 1-D (in which case its length must be the size of *a* along the given axis) or of the same shape as *a*. If *weights=None*, then all data in *a* are assumed to have a weight equal to one.

returned [bool, optional] Default is *False*. If *True*, the tuple (*average*, *sum_of_weights*) is returned, otherwise only the average is returned. If *weights=None*, *sum_of_weights* is equivalent to the number of elements over which the average is taken.

average, [sum_of_weights] [tensor_type or double] Return the average along the specified axis. When returned is *True*, return a tuple with the average as the first element and the sum of the weights as the second element. The return type is *Float* if *a* is of integer type, otherwise it is of the same type as *a*. *sum_of_weights* is of the same type as *average*.

ZeroDivisionError When all weights along axis are zero. See *numpy.ma.average* for a version robust to this type of error.

TypeError When the length of 1D *weights* is not the same as the shape of *a* along axis.

mean

```
>>> import mars.tensor as mt
```

```
>>> data = list(range(1,5))
>>> data
[1, 2, 3, 4]
>>> mt.average(data).execute()
2.5
```

(continues on next page)

(continued from previous page)

```
>>> mt.average(range(1,11), weights=range(10,0,-1)).execute()
4.0
```

```
>>> data = mt.arange(6).reshape((3,2))
>>> data.execute()
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> mt.average(data, axis=1, weights=[1./4, 3./4]).execute()
array([ 0.75,  2.75,  4.75])
>>> mt.average(data, weights=[1./4, 3./4]).execute()
Traceback (most recent call last):
...
TypeError: Axis must be specified when shapes of a and weights differ.
```

mars.tensor.mean

mars.tensor.**mean**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=None*, *combine_size=None*)

Compute the arithmetic mean along the specified axis.

Returns the average of the array elements. The average is taken over the flattened tensor by default, otherwise over the specified axis. *float64* intermediate and return values are used for integer inputs.

a [array_like] Tensor containing numbers whose mean is desired. If *a* is not an tensor, a conversion is attempted.

axis [None or int or tuple of ints, optional] Axis or axes along which the means are computed. The default is to compute the mean of the flattened array.

If this is a tuple of ints, a mean is performed over multiple axes, instead of a single axis or all the axes as before.

dtype [data-type, optional] Type to use in computing the mean. For integer inputs, the default is *float64*; for floating point inputs, it is the same as the input dtype.

out [Tensor, optional] Alternate output tensor in which to place the result. The default is `None`; if provided, it must have the same shape as the expected output, but the type will be cast if necessary. See *doc.ufuncs* for details.

keepdims [bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input tensor.

If the default value is passed, then *keepdims* will not be passed through to the *mean* method of sub-classes of *Tensor*, however any non-default value will be. If the sub-classes *sum* method does not implement *keepdims* any exceptions will be raised.

combine_size: int, optional The number of chunks to combine.

m [Tensor, see dtype parameter above] If *out=None*, returns a new tensor containing the mean values, otherwise a reference to the output array is returned.

average : Weighted average std, var, nanmean, nanstd, nanvar

The arithmetic mean is the sum of the elements along the axis divided by the number of elements.

Note that for floating-point input, the mean is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for *float32* (see example below). Specifying a higher-precision accumulator using the *dtype* keyword can alleviate this issue.

By default, *float16* results are computed using *float32* intermediates for extra precision.

```
>>> import mars.tensor as mt
```

```
>>> a = mt.array([[1, 2], [3, 4]])
>>> mt.mean(a).execute()
2.5
>>> mt.mean(a, axis=0).execute()
array([ 2.,  3.])
>>> mt.mean(a, axis=1).execute()
array([ 1.5,  3.5])
```

In single precision, *mean* can be inaccurate:

```
>>> a = mt.zeros((2, 512*512), dtype=mt.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> mt.mean(a).execute()
0.54999924
```

Computing the mean in float64 is more accurate:

```
>>> mt.mean(a, dtype=mt.float64).execute()
0.55000000074505806
```

mars.tensor.std

`mars.tensor.`**`std`**(*a*, *axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=None*, *combine_size=None*)

Compute the standard deviation along the specified axis.

Returns the standard deviation, a measure of the spread of a distribution, of the tensor elements. The standard deviation is computed for the flattened tensor by default, otherwise over the specified axis.

a [array_like] Calculate the standard deviation of these values.

axis [None or int or tuple of ints, optional] Axis or axes along which the standard deviation is computed. The default is to compute the standard deviation of the flattened tensor.

If this is a tuple of ints, a standard deviation is performed over multiple axes, instead of a single axis or all the axes as before.

dtype [dtype, optional] Type to use in computing the standard deviation. For tensors of integer type the default is float64, for tensors of float types it is the same as the array type.

out [Tensor, optional] Alternative output tensor in which to place the result. It must have the same shape as the expected output but the type (of the calculated values) will be cast if necessary.

ddof [int, optional] Means Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of elements. By default *ddof* is zero.

keepdims [bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input tensor.

If the default value is passed, then *keepdims* will not be passed through to the *std* method of sub-classes of *Tensor*, however any non-default value will be. If the sub-classes *sum* method does not implement *keepdims* any exceptions will be raised.

combine_size: int, optional The number of chunks to combine.

standard_deviation [Tensor, see dtype parameter above.] If *out* is None, return a new tensor containing the standard deviation, otherwise return a reference to the output array.

var, mean, nanmean, nanstd, nanvar

The standard deviation is the square root of the average of the squared deviations from the mean, i.e., `std = sqrt(mean(abs(x - x.mean())**2))`.

The average squared deviation is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead. In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of the infinite population. `ddof=0` provides a maximum likelihood estimate of the variance for normally distributed variables. The standard deviation computed in this function is the square root of the estimated variance, so even with `ddof=1`, it will not be an unbiased estimate of the standard deviation per se.

Note that, for complex numbers, *std* takes the absolute value before squaring, so that the result is always real and nonnegative.

For floating-point input, the *std* is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the *dtype* keyword can alleviate this issue.

```
>>> import mars.tensor as mt
```

```
>>> a = mt.array([[1, 2], [3, 4]])
>>> mt.std(a).execute()
1.1180339887498949
>>> mt.std(a, axis=0).execute()
array([ 1.,  1.])
>>> mt.std(a, axis=1).execute()
array([ 0.5,  0.5])
```

In single precision, std() can be inaccurate:

```
>>> a = mt.zeros((2, 512*512), dtype=mt.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> mt.std(a).execute()
0.45000005
```

Computing the standard deviation in float64 is more accurate:

```
>>> mt.std(a, dtype=mt.float64).execute()
0.44999999925494177
```

mars.tensor.var

`mars.tensor.var`(*a*, *axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=None*, *combine_size=None*)
Compute the variance along the specified axis.

Returns the variance of the tensor elements, a measure of the spread of a distribution. The variance is computed for the flattened tensor by default, otherwise over the specified axis.

a [array_like] Tensor containing numbers whose variance is desired. If *a* is not a tensor, a conversion is attempted.

axis [None or int or tuple of ints, optional] Axis or axes along which the variance is computed. The default is to compute the variance of the flattened array.

If this is a tuple of ints, a variance is performed over multiple axes, instead of a single axis or all the axes as before.

dtype [data-type, optional] Type to use in computing the variance. For arrays of integer type the default is *float32*; for tensors of float types it is the same as the tensor type.

out [Tensor, optional] Alternate output array in which to place the result. It must have the same shape as the expected output, but the type is cast if necessary.

ddof [int, optional] "Delta Degrees of Freedom": the divisor used in the calculation is `N - ddof`, where `N` represents the number of elements. By default *ddof* is zero.

keepdims [bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input tensor.

If the default value is passed, then *keepdims* will not be passed through to the *var* method of sub-classes of *Tensor*, however any non-default value will be. If the sub-classes *sum* method does not implement *keepdims* any exceptions will be raised.

combine_size: int, optional The number of chunks to combine.

variance [Tensor, see dtype parameter above] If `out=None`, returns a new tensor containing the variance; otherwise, a reference to the output tensor is returned.

std , mean, nanmean, nanstd, nanvar

The variance is the average of the squared deviations from the mean, i.e., `var = mean(abs(x - x.mean())**2)`.

The mean is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead. In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of a hypothetical infinite population. `ddof=0` provides a maximum likelihood estimate of the variance for normally distributed variables.

Note that for complex numbers, the absolute value is taken before squaring, so that the result is always real and nonnegative.

For floating-point input, the variance is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for *float32* (see example below). Specifying a higher-accuracy accumulator using the `dtype` keyword can alleviate this issue.

```
>>> import mars.tensor as mt
```

```
>>> a = mt.array([[1, 2], [3, 4]])
>>> mt.var(a).execute()
1.25
>>> mt.var(a, axis=0).execute()
array([ 1.,  1.])
>>> mt.var(a, axis=1).execute()
array([ 0.25,  0.25])
```

In single precision, var() can be inaccurate:

```
>>> a = mt.zeros((2, 512*512), dtype=mt.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> mt.var(a).execute()
0.20250003
```

Computing the variance in float64 is more accurate:

```
>>> mt.var(a, dtype=mt.float64).execute()
0.20249999932944759
>>> ((1-0.55)**2 + (0.1-0.55)**2)/2
0.2025
```

mars.tensor.nanmean

mars.tensor.**nanmean**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=None*, *combine_size=None*)

Compute the arithmetic mean along the specified axis, ignoring NaNs.

Returns the average of the tensor elements. The average is taken over the flattened tensor by default, otherwise over the specified axis. *float64* intermediate and return values are used for integer inputs.

For all-NaN slices, NaN is returned and a *RuntimeWarning* is raised.

a [array_like] Tensor containing numbers whose mean is desired. If *a* is not an tensor, a conversion is attempted.

axis [int, optional] Axis along which the means are computed. The default is to compute the mean of the flattened tensor.

dtype [data-type, optional] Type to use in computing the mean. For integer inputs, the default is *float64*; for inexact inputs, it is the same as the input dtype.

out [Tensor, optional] Alternate output tensor in which to place the result. The default is `None`; if provided, it must have the same shape as the expected output, but the type will be cast if necessary. See *doc.ufuncs* for details.

keepdims [bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *a*.

If the value is anything but the default, then *keepdims* will be passed through to the *mean* or *sum* methods of sub-classes of *Tensor*. If the sub-classes methods does not implement *keepdims* any exceptions will be raised.

combine_size: int, optional The number of chunks to combine.

m [Tensor, see dtype parameter above] If *out=None*, returns a new array containing the mean values, otherwise a reference to the output array is returned. Nan is returned for slices that contain only NaNs.

average : Weighted average mean : Arithmetic mean taken while not ignoring NaNs var, nanvar

The arithmetic mean is the sum of the non-NaN elements along the axis divided by the number of non-NaN elements.

Note that for floating-point input, the mean is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for *float32*. Specifying a higher-precision accumulator using the *dtype* keyword can alleviate this issue.

```
>>> import mars.tensor as mt
```

```
>>> a = mt.array([[1, mt.nan], [3, 4]])
>>> mt.nanmean(a).execute()
2.6666666666666665
>>> mt.nanmean(a, axis=0).execute()
array([ 2.,  4.])
>>> mt.nanmean(a, axis=1).execute()
array([ 1.,  3.5])
```

mars.tensor.nanstd

mars.tensor.**nanstd**(*a*, *axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=None*, *combine_size=None*)

Compute the standard deviation along the specified axis, while ignoring NaNs.

Returns the standard deviation, a measure of the spread of a distribution, of the non-NaN tensor elements. The standard deviation is computed for the flattened tensor by default, otherwise over the specified axis.

For all-NaN slices or slices with zero degrees of freedom, NaN is returned and a *RuntimeWarning* is raised.

a [array_like] Calculate the standard deviation of the non-NaN values.

axis [int, optional] Axis along which the standard deviation is computed. The default is to compute the standard deviation of the flattened tensor.

dtype [dtype, optional] Type to use in computing the standard deviation. For tensors of integer type the default is float64, for tensors of float types it is the same as the tensor type.

out [Tensor, optional] Alternative output tensor in which to place the result. It must have the same shape as the expected output but the type (of the calculated values) will be cast if necessary.

ddof [int, optional] Means Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of non-NaN elements. By default *ddof* is zero.

keepdims [bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *a*.

If this value is anything but the default it is passed through as-is to the relevant functions of the sub-classes. If these functions do not have a *keepdims* kwarg, a RuntimeError will be raised.

combine_size: int, optional The number of chunks to combine.

standard_deviation [ndarray, see dtype parameter above.] If *out* is None, return a new array containing the standard deviation, otherwise return a reference to the output tensor. If ddof is >= the number of non-NaN elements in a slice or the slice contains only NaNs, then the result for that slice is NaN.

var, mean, std nanvar, nanmean

The standard deviation is the square root of the average of the squared deviations from the mean: `std = sqrt(mean(abs(x - x.mean())**2))`.

The average squared deviation is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead. In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of the infinite population. `ddof=0` provides a maximum likelihood estimate of the variance for normally distributed variables. The standard deviation computed in this function is the square root of the estimated variance, so even with `ddof=1`, it will not be an unbiased estimate of the standard deviation per se.

Note that, for complex numbers, *std* takes the absolute value before squaring, so that the result is always real and nonnegative.

For floating-point input, the *std* is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the *dtype* keyword can alleviate this issue.

```
>>> import mars.tensor as mt
```

```
>>> a = mt.array([[1, mt.nan], [3, 4]])
>>> mt.nanstd(a).execute()
1.247219128924647
>>> mt.nanstd(a, axis=0).execute()
array([ 1.,  0.])
>>> mt.nanstd(a, axis=1).execute()
array([ 0.,  0.5])
```

mars.tensor.nanvar

mars.tensor.**nanvar**(*a*, *axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=None*, *combine_size=None*)

Compute the variance along the specified axis, while ignoring NaNs.

Returns the variance of the tensor elements, a measure of the spread of a distribution. The variance is computed for the flattened tensor by default, otherwise over the specified axis.

For all-NaN slices or slices with zero degrees of freedom, NaN is returned and a *RuntimeWarning* is raised.

a [array_like] Tensor containing numbers whose variance is desired. If *a* is not a tensor, a conversion is attempted.

axis [int, optional] Axis along which the variance is computed. The default is to compute the variance of the flattened array.

dtype [data-type, optional] Type to use in computing the variance. For tensors of integer type the default is *float32*; for tensors of float types it is the same as the tensor type.

out [Tensor, optional] Alternate output tensor in which to place the result. It must have the same shape as the expected output, but the type is cast if necessary.

ddof [int, optional] "Delta Degrees of Freedom": the divisor used in the calculation is `N - ddof`, where `N` represents the number of non-NaN elements. By default *ddof* is zero.

keepdims [bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *a*.

combine_size: int, optional The number of chunks to combine.

variance [Tensor, see dtype parameter above] If *out* is None, return a new tensor containing the variance, otherwise return a reference to the output tensor. If ddof is >= the number of non-NaN elements in a slice or the slice contains only NaNs, then the result for that slice is NaN.

std : Standard deviation mean : Average var : Variance while not ignoring NaNs nanstd, nanmean

The variance is the average of the squared deviations from the mean, i.e., `var = mean(abs(x - x.mean())**2)`.

The mean is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead. In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of a hypothetical infinite population. `ddof=0` provides a maximum likelihood estimate of the variance for normally distributed variables.

Note that for complex numbers, the absolute value is taken before squaring, so that the result is always real and nonnegative.

For floating-point input, the variance is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for *float32* (see example below). Specifying a higher-accuracy accumulator using the `dtype` keyword can alleviate this issue.

For this function to work on sub-classes of Tensor, they must define *sum* with the kwarg *keepdims*

```
>>> import mars.tensor as mt
```

```
>>> a = mt.array([[1, mt.nan], [3, 4]])
>>> mt.nanvar(a).execute()
1.5555555555555554
>>> mt.nanvar(a, axis=0).execute()
array([ 1.,  0.])
>>> mt.nanvar(a, axis=1).execute()
array([ 0.,  0.25])
```

Correlating

mars.tensor.corrcoef	Return Pearson product-moment correlation coefficients.
mars.tensor.cov	Estimate a covariance matrix, given data and weights.

mars.tensor.corrcoef

mars.tensor.**corrcoef**(*x*, *y=None*, *rowvar=True*)

Return Pearson product-moment correlation coefficients.

Please refer to the documentation for *cov* for more detail. The relationship between the correlation coefficient matrix, *R*, and the covariance matrix, *C*, is

$$R_{ij} = \frac{C_{ij}}{\sqrt{C_{ii} * C_{jj}}}$$

The values of *R* are between -1 and 1, inclusive.

x [array_like] A 1-D or 2-D array containing multiple variables and observations. Each row of *x* represents a variable, and each column a single observation of all those variables. Also see *rowvar* below.

y [array_like, optional] An additional set of variables and observations. *y* has the same shape as *x*.

rowvar [bool, optional] If *rowvar* is True (default), then each row represents a variable, with observations in the columns. Otherwise, the relationship is transposed: each column represents a variable, while the rows contain observations.

R [Tensor] The correlation coefficient matrix of the variables.

cov : Covariance matrix

Due to floating point rounding the resulting tensor may not be Hermitian, the diagonal elements may not be 1, and the elements may not satisfy the inequality abs(a) <= 1. The real and imaginary parts are clipped to the interval [-1, 1] in an attempt to improve on that situation but is not much help in the complex case.

This function accepts but discards arguments *bias* and *ddof*. This is for backwards compatibility with previous versions of this function. These arguments had no effect on the return values of the function and can be safely ignored in this and previous versions of numpy.

mars.tensor.cov

mars.tensor.**cov**(*m*, *y=None*, *rowvar=True*, *bias=False*, *ddof=None*, *fweights=None*, *aweights=None*)

Estimate a covariance matrix, given data and weights.

Covariance indicates the level to which two variables vary together. If we examine N-dimensional samples, $X = [x_1, x_2, ...x_N]^T$, then the covariance matrix element C_{ij} is the covariance of x_i and x_j. The element C_{ii} is the variance of x_i.

See the notes for an outline of the algorithm.

m [array_like] A 1-D or 2-D array containing multiple variables and observations. Each row of *m* represents a variable, and each column a single observation of all those variables. Also see *rowvar* below.

y [array_like, optional] An additional set of variables and observations. *y* has the same form as that of *m*.

rowvar [bool, optional] If *rowvar* is True (default), then each row represents a variable, with observations in the columns. Otherwise, the relationship is transposed: each column represents a variable, while the rows contain observations.

bias [bool, optional] Default normalization (False) is by `(N - 1)`, where `N` is the number of observations given (unbiased estimate). If *bias* is True, then normalization is by `N`. These values can be overridden by using the keyword `ddof` in numpy versions >= 1.5.

ddof [int, optional] If not `None` the default value implied by *bias* is overridden. Note that `ddof=1` will return the unbiased estimate, even if both *fweights* and *aweights* are specified, and `ddof=0` will return the simple average. See the notes for the details. The default value is `None`.

fweights [array_like, int, optional] 1-D tensor of integer freguency weights; the number of times each observation vector should be repeated.

aweights [array_like, optional] 1-D tensor of observation vector weights. These relative weights are typically large for observations considered "important" and smaller for observations considered less "important". If `ddof=0` the array of weights can be used to assign probabilities to observation vectors.

out [Tensor] The covariance matrix of the variables.

corrcoef : Normalized covariance matrix

Assume that the observations are in the columns of the observation array *m* and let `f = fweights` and `a = aweights` for brevity. The steps to compute the weighted covariance are as follows:

```
>>> w = f * a
>>> v1 = mt.sum(w)
>>> v2 = mt.sum(w * a)
>>> m -= mt.sum(m * w, axis=1, keepdims=True) / v1
>>> cov = mt.dot(m * w, m.T) * v1 / (v1**2 - ddof * v2)
```

Note that when `a == 1`, the normalization factor `v1 / (v1**2 - ddof * v2)` goes over to `1 / (np.sum(f) - ddof)` as it should.

Consider two variables, x_0 and x_1, which correlate perfectly, but in opposite directions:

```
>>> import mars.tensor as mt
```

```
>>> x = mt.array([[0, 2], [1, 1], [2, 0]]).T
>>> x.execute()
array([[0, 1, 2],
       [2, 1, 0]])
```

Note how x_0 increases while x_1 decreases. The covariance matrix shows this clearly:

```
>>> mt.cov(x).execute()
array([[ 1., -1.],
       [-1.,  1.]])
```

Note that element $C_{0,1}$, which shows the correlation between x_0 and x_1, is negative.

Further, note how *x* and *y* are combined:

```
>>> x = [-2.1, -1,  4.3]
>>> y = [3,  1.1,  0.12]
>>> X = mt.stack((x, y), axis=0)
>>> print(mt.cov(X).execute())
[[ 11.71        -4.286     ]
 [ -4.286        2.14413333]]
>>> print(mt.cov(x, y).execute())
[[ 11.71        -4.286     ]
 [ -4.286        2.14413333]]
>>> print(mt.cov(x).execute())
11.71
```

Histograms

mars.tensor.digitize	Return the indices of the bins to which each value in input tensor belongs.

mars.tensor.digitize

mars.tensor.**digitize**(*x*, *bins*, *right=False*)

Return the indices of the bins to which each value in input tensor belongs.

Each index `i` returned is such that `bins[i-1] <= x < bins[i]` if *bins* is monotonically increasing, or `bins[i-1] > x >= bins[i]` if *bins* is monotonically decreasing. If values in *x* are beyond the bounds of *bins*, 0 or `len(bins)` is returned as appropriate. If right is True, then the right bin is closed so that the index `i` is such that `bins[i-1] < x <= bins[i]` or `bins[i-1] >= x > bins[i]` if *bins* is monotonically increasing or decreasing, respectively.

x [array_like] Input tensor to be binned.

bins [array_like] Array of bins. It has to be 1-dimensional and monotonic.

right [bool, optional] Indicating whether the intervals include the right or the left bin edge. Default behavior is (right==False) indicating that the interval does not include the right edge. The left bin end is open in this case, i.e., bins[i-1] <= x < bins[i] is the default behavior for monotonically increasing bins.

out [Tensor of ints] Output tensor of indices, of same shape as *x*.

ValueError If *bins* is not monotonic.

TypeError If the type of the input is complex.

bincount, histogram, unique, searchsorted

If values in *x* are such that they fall outside the bin range, attempting to index *bins* with the indices that *digitize* returns will result in an IndexError.

mt.digitize is implemented in terms of *mt.searchsorted*. This means that a binary search is used to bin the values, which scales much better for larger number of bins than the previous linear search. It also removes the requirement for the input array to be 1-dimensional.

```
>>> import mars.tensor as mt
```

```
>>> x = mt.array([0.2, 6.4, 3.0, 1.6])
>>> bins = mt.array([0.0, 1.0, 2.5, 4.0, 10.0])
>>> inds = mt.digitize(x, bins)
>>> inds.execute()
array([1, 4, 3, 2])
```

```
>>> x = mt.array([1.2, 10.0, 12.4, 15.5, 20.])
>>> bins = mt.array([0, 5, 10, 15, 20])
>>> mt.digitize(x,bins,right=True).execute()
array([1, 2, 3, 4, 4])
>>> mt.digitize(x,bins,right=False).execute()
array([1, 3, 3, 4, 5])
```

2.7 Sparse tensor

Mars tensor supports sparse tensor, unfortunately, only 2-D sparse tensors are available for now. Multi-dimensional sparse tensors will be supported later.

2.7.1 Functions to create sparse tensor

mars.tensor.tensor	
mars.tensor.zeros	Return a new tensor of given shape and type, filled with zeros.
mars.tensor.eye	Return a 2-D tensor with ones on the diagonal and zeros elsewhere.
mars.tensor.identity	Return the identity tensor.
mars.tensor.random.randint	Return random integers from *low* (inclusive) to *high* (exclusive).

2.8 Local Execution

When *eager mode* is not enabled, which is the default behavior, Mars tensor will not be executed unless users call `execute` or `session.run` methods.

If no session is created explicitly, the `execute` will create a local session, and mark it as a default session.

2.8.1 Session

Users can create a new session by `new_session` method, if no argument is provided, a local session will be generated.

```
>>> from mars.session import new_session

>>> sess = new_session()  # create a session
```

By calling `as_default` of a session, the session will be marked as the default session.

```
>>> sess.as_default()
```

More than one mars tensors can be passed to `session.run`, and calculate the results for each tensor.

```
>>> a = mt.ones((5, 5), chunk_size=3)
>>> b = a + 1
>>> c = a * 4
>>> sess.run(b, c)
(array([[2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2.]]), array([[4., 4., 4., 4., 4.],
       [4., 4., 4., 4., 4.],
       [4., 4., 4., 4., 4.],
       [4., 4., 4., 4., 4.],
       [4., 4., 4., 4., 4.]]))
```

2.8.2 Execute a tensor

For a single tensor, `execute` can be called.

```
>>> a = mt.random.rand(3, 4)
>>> a.sum().execute()
7.0293719034458455
```

Session can be specified by the argument `session`.

```
>>> a.sum().execute(session=sess)
6.12833989477539
```

2.9 Eager Mode

Note: New in version 0.2.0a2

Mars supports eager mode which makes it friendly for developing and easy to debug.

Users can enable the eager mode by options, set options at the beginning of the program or console session.

```
>>> from mars.config import options
>>> options.eager_mode = True
```

Or use a context.

```
>>> from mars.config import option_context

>>> with option_context() as options:
>>>     options.eager_mode = True
>>>     # the eager mode is on only for the with statement
>>>     ...
```

If eager mode is on, tensor will be executed immediately by default session once it is created.

```
>>> import mars.tensor as mt
>>> from mars.config import options
>>> options.eager_mode = True
>>> t = mt.arange(6).reshape((2, 3))
>>> print(t)
Tensor(op=TensorRand, shape=(2, 3), data=
[[0 1 2]
[3 4 5]])
```

Use `fetch` to obtain numpy value from a tensor:

```
>>> t.fetch()
array([[0, 1, 2],
       [3, 4, 5]])
```

2.10 Architecture

Mars provides a library for distributed execution of tensors. The distributed applications are built with actor model provided by `mars.actors` and consists of three parts: the scheduler, the worker and the web service.

Users submit their tasks in graphs built with tensors. The web service accepts tensor graphs and sends them into a scheduler, where graphs are compiled into operand graphs, analyzed and partitioned before submitted to workers. The scheduler then creates and scatters operand actors who control task execution on workers on other schedulers given consistent hashing. Then operands are activated and executed in topological order. When all operands related to terminating tensors are executed, the graph will be marked as finished and the client can pull the result from workers, proxied by the scheduler. The whole procedure can be seen in the graph below.

2.10.1 Job Submission

Jobs are submitted into Mars via RESTful APIs. Users type tensor operations and run a tensor by calling `session.run(tensor)`, which builds a tensor graph given the operations created by the user. This graph is sent to the web api and a GraphActor is created given consistent hashing in the cluster to handle the tensor graph. After that the web client begins querying the state of the graph until termination.

In the GraphActor, we first convert the tensor graph into an operand graph via tiling methods. This enables the graph to run in parallel. After that, several analyzes are performed on the graph to obtain operand priorities and assign workers for the operand, which can be seen in detail in *graph preparation* and *scheduling policy* section. Then OperandActor is created for every operand to control detailed execution. When an operand is in `READY` state, as described in *operand states* section, a worker will be selected and the operand is submitted into the worker for execution.

2.10.2 Execution Control

When an operand is submitted to a worker, the OperandActor on the scheduler listen to its callback. When the execution is successful, successors of that operand will be scheduled. When the execution failed, the OperandActor will retry several times before announcing the execution as fatal.

2.10.3 Job Cancellation

Users can cancel a running job via RESTful API. The request is written into state storage first and then called in GraphActor. If the graph is under preparation, it will stop immediately when the stop request is detected in state storage. Otherwise every operand is scanned and the states will be set as `CANCELLING`. When the operand is currently running, a stop request will be sent into workers which results in `ExecutionInterrupted` exception in workers. When this exception is received in OperandActor, the state of the operand will be marked as `CANCELLED`.

2.11 Graph Preparation

When a tensor graph is submitted into Mars scheduler, a graph comprises of operands and chunks will be generated given `chunk_size` parameters passed in data sources.

2.11.1 Graph Compose

After tiling a tensor graph into a chunk graph, we will combine adjacent nodes to reduce graph size as well as to utilize acceleration libraries such as `numexpr`. Currently Mars only merges operands that forms a single chain without branches. For example, when executing code

```
import mars.tensor as mt
a = mt.random.rand(100, chunk_size=100)
b = mt.random.rand(100, chunk_size=100)
c = (a + b).sum()
```

Mars will compose operand ADD and SUM into one FUSE node. RAND operands are excluded because they don't form a line with ADD and SUM.

2.11.2 Initial Worker Assignment

Assigning operands to workers are crucial to the performance of graph execution. Random worker assignment will contribute to huge network cost and imbalanced workload between different workers. Since the workers of non-initial operands can be effectively decided given data distribution and cluster idleness, we only assign workers for initial nodes in graph preparation stage.

Initial worker assignment should obey several principles. First, the number of operands assigned to each worker should be balanced. This makes full use of the cluster especially in the late stage of graph execution. Secondly, operand assignment should minify the amount of network transfer in its descendants. That is, locality need to be observed in the assignment process.

Note that these principles sometimes collides with each other. That is, a network-minimal solution may be quite biased. We developed a heuristic algorithm in practice that takes a balance between minimal network transfer and worker load balance. The algorithm is described below:

1. Select the first worker who does not have any operands;

2. Start breadth-first search on the undirected graph produced from the operand graph;
3. When an initial operand is visited, we assign it to the worker we selected in Step 1;
4. Stop assignment when the number of operands visited is greater than the average number of operands for every worker;
5. Go to Step 1 when there are workers left.

2.12 Scheduling Policy

When an operand graph is being executed, proper selection of execution order will reduce total amount of data stored in the cluster, thus reducing the probability that chunks are spilled into disks. Proper selection of workers can also reduce the amount of data needed to transfer in execution.

2.12.1 Operand Selection

Proper execution order can significantly reduce the number of objects stored in the cluster. We show the example of tree reduction in the graph below, where ovals represent operands and rectangles represent chunks. Red color means that the operand is being executed, and blue color means that the operand is ready for execution. Green color means that the chunk is stored, while the gray color means that chunks or operands are freed. Assume that we have 2 workers, and work load of all operands are the same. Both graphs show one operand selection strategy that is executed after 5 time unit. The left graph show the scenario when nodes are executed in hierarchical order, while the right show that the graph is executed in depth-first order. The strategy on the left graph leaves 6 chunks stored in the cluster, while the right only 2.

Given that our goal is to reduce the amount of data stored in the cluster during execution, we put a priority for operands when they are ready for execution:

1. The operand with greater depth shall be executed earlier;
2. The operand required by deeper operands shall be executed earlier;
3. The operand with smaller output size shall be executed first.

2.12.2 Worker Selection

The worker of initial operands are decided when the scheduler prepares an operand graph. We choose the worker of descendant operands given the location of input chunks. When there are multiple workers providing minimal network transfer, a worker satisfying resource requirements are selected.

2.13 Operand States

Every operand in Mars is scheduled independently by an OperandActor. The execution is designed as a state transition process. We assign a state handling function for every state to control the execution procedure. Every operand is at `UNSCHEDULED` state when the actor is initially initialized. When certain conditions are met, the operand switches into another state and perform corresponding actions. If an operand is recovered from KV store, its state when scheduler crashes will be loaded and the state is resumed. The state transition graph can be shown below:

We illustrate the meaning of every state and actions Mars take when entering these states.

- UNSCHEDUED: the operand is in this state when it is not ready to execute.
- READY: the operand is in this state when all input chunks are ready. When this state is entered, the OperandActor submits the operand to all workers selected in AssignerActor. If the operand is about to run in one of the selected workers, it will respond to the scheduler and the scheduler suspends the operand on other workers and start executing the operand on the requesting worker.
- RUNNING: the operand is in this state when it is assigned to a worker or already started execution. When this state is entered, the OperandActor checks if it has been submitted to the worker. If not submitted, the operand is built into an "executable dag" containing FetchChunks. Then a callback is registered in the worker to handle execution finish.
- FINISHED: the operand is in this state when the operand finishes execution. When this state is entered, a terminal operand will send a notification to its GraphActor to decide if the whole graph finishes execution. What's more, the OperandActor looks for precedent and successor chunks and notify them. When a predecessor receives the notification, it checks if all its successors finishes execution. If so, the data of that operand can be freed. When a successor receives the notification, it checks if all of its predecessors are finished. If so, the operand itself can move to READY.
- FREED: the operand is in this state when all data related to this operand is freed.
- FATAL: the operand is in this state when itself or some predecessor failed to execute. When this state is entered, the OperandActor try to pass this state down to its successors.
- CANCELLING: the operand is in this state when it is being cancelled. If the operand is previously running, a cancel request will be sent to the worker.
- CANCELLED: the operand is in this state when it is cancelled and stops running. When this state is entered, the OperandActor tries to switch its descendants into CANCELLING.

2.14 Execution in Worker

A Mars worker consists of multiple processes to reduce the impact of the notorious global interpreter lock (GIL) in Python. Executions run in separate processes. To reduce unnecessary memory copy and inter-process communication, shared memory is used to store computation results.

When an operand is being executed in a worker, it will first allocate memory. Then data from other workers or from files already spilled to disk are loaded. After that all data required are in memory and calculation can start. When calculation is done, the worker then put the result into shared memory cache. These four states can be seen in the graph below.

2.14.1 Execution Control

A Mars worker starts an ExecutionActor to control **all** the operands running on the worker. It does not actually do calculation or data transfer itself, but submit these actions to other actors.

OperandActors in schedulers submit an operand into workers through `execute_graph` calls. Then a callback is registered via `add_finish_callback`. This design allows finish message be sent to different places, which is necessary for failover.

ExecutionActor uses `mars.promise` module to handle multiple operands simultaneously. Execution steps are chained via `then` method of the `Promise` class. When the final result is successfully stored, all registered callbacks will be invoked. When exception raises in any chained promise, the final exception handler registered with `catch` will try handling this exception.

2.14.2 Operand Ordering

All operands in `READY` state are submitted into workers selected by the scheduler. Therefore, the number of operands submitted to the worker is beyond the capacity of the worker most of the time during execution, and the worker need to sort these operands in order before picking some of the operands for execution. This is done in TaskQueueActor in worker, where a priority queue is maintained to store information of the operands. An allocator runs periodically, trying to allocate resources for the operand at the head of the queue till no free space left. The allocator is also triggered when a new operand arrives, or an operand finishes execution.

2.14.3 Memory Management

Mars worker manages two different parts of memory. The first is private memory in every worker process, handled by every worker process. The second is shared memory between all worker processes, handled by plasma_store in Apache Arrow.

To avoid out-of-memory error in process memory, we introduce a worker-level QuotaActor to allocate process memory. Before an operand starts execution, it sends a memory batch request to the QuotaActor, asking for memory blocks for its input and output chunks. When memory quota left can satisfy the request, the QuotaActor accepts the request. Otherwise the request is queued. After the memory block is released, the allocation is freed and QuotaActor can accept other requests.

Shared memory is handled by plasma_store, which often takes of up to 50% of total memory. As there is no risk of out-of-memory, this part of memory is allocated directly without quota requests. When shared memory is exhausted, Mars worker tries to spill unused chunks into disk.

Chunks spill into disks may be used by later operands, and loading data from disk into shared memory can be costly in IO, especially when the shared memory is exhausted, and we need to spill other chunks to hold the loaded chunk. Therefore, when data sharing is not needed, for instance, the input chunk is only used by a single operand, it can be loaded into private memory instead of shared memory. This can significantly reduce execution time.

2.15 Fault Tolerance

Currently Mars supports two levels of fault tolerance: process-level and worker-level. Scheduler-level support is not implemented now.

2.15.1 Process-level Fault Tolerance

Mars uses multiple processes in its worker. When a worker process fails, for instance, gets killed because of out of memory, and the process is not Process 0 where control actors run, Mars worker will mark relevant tasks as failed, start another process, restore actors on it and the retry mechanism will restart the failed task.

2.15.2 Worker-level Fault Tolerance

Note: New in version 0.2.0a2

As Mars uses execution graphs to schedule tasks, when some workers fail, scheduler will find lost chunks and work out affected operands. After that the spotted operands are rescheduled.

Failure Notification and Processing

When a worker fails to respond, actors both in other workers and schedulers will detect it and send a feedback to ResourceActor, where changes in worker list is accepted and broadcast into all the sessions. When SessionActors accept the message, they will collect keys of missing data and relay all collected information to running GraphActors, where fail-over decisions are actually made.

The failure notification procedure is illustrated in the graph below.

When accepting a fail-over call, a GraphActor will first try reassigning states of affected operands, reassigning initial workers for initial operands, and then send updates to corresponding OperandActors to rerun these operands.

Reassigning States

When some workers fail, data stored in these workers are lost. Therefore we need to change the states of these operands in order to run them again. As is stated in *operand states*, data generated by an operand only exist when operands are in `FINISHED` state, we perform a two-pass scanning procedure to calculate new assignments for operands:

1. Scan all FINISHED operands whose data are lost and mark them and their successors as affected;
2. Scan the graph from bottom to top, starting from affected operands;
3. For every affected operand, we scan its predecessors. If the data of the predecessor is lost or in a state without data or generating data, for instance, `FREED` or `UNSCHEDULED`, the predecessor will be marked as affected;
4. Mark current operand as `READY` when no predecessors are affected, otherwise it will be marked as `UNSCHEDULED`;
5. When there are no affected operands to be scanned, stop, otherwise go to Step 2.

Reassigning Initial Workers

When workers fail, some of initial operands assigned in *graph preparation* step may not have workers to execute on. What's more, change in number of workers may lead to imbalance in worker assignments. We solve this problem by applying an adaptive worker reassigning strategy. The algorithm framework is similar with the original one except that we do not visit initial operands which are executed, and the stop criterion of visiting operands in Step 3 is now limited to the average number of operands per worker minus the number of operands already executed or determined to run in the candidate worker.

2.16 Contributing to Mars

Mars is an open-sourced project released under Apache License 2.0. We welcome and thanks for your contributions. Here are some guidelines you may find useful when you want to make some change of Mars.

2.16.1 General Guidelines

Mars hosts and maintains its code on Github. We provide a generalized guide for opening issues and pull requests.

2.16.2 Setting Up a Development Environment

Unless you want to develop or debug Mars under Windows, it is strongly recommended to develop Mars under MacOS or Linux, where you can test all functions of Mars. The steps listed below are applicable on MacOS and Linux.

Install in Conda

It is recommended to develop Mars with conda. When you want ot install Mars for development, use the following steps to create an environment and install Mars inside it:

```
git clone https://github.com/mars-project/mars.git
cd mars
conda create -n mars-dev --file conda-spec.txt python=3.7
source activate mars-dev
conda install -c conda-forge pyarrow tiledb-py
pip install -e .
```

Change `3.7` into the version of Python you want to install, and `mars-dev` into your preferred environment name.

Other Python Distributions

Mars has a `dev` option for installation. When you want to install Mars for development, use the following steps:

```
pip install --upgrade setuptools pip
git clone https://github.com/mars-project/mars.git
cd mars
pip install -e ".[dev]"
```

If you are using a system-wide Python installation and you only want to install Mars for you, you can add `--user` to the `pip install` commands.

Verification

After installing Mars, you can check if Mars is installed correctly by running

```
python -c "import mars; print(mars.__version__)"
```

If this command correctly outputs your Mars version, Mars is correctly installed.

Rebuilding Cython Code

Mars uses Cython to accelerate part of its code. After you change Cython source code, you need to compile them into binaries by executing the command below on the root of Mars project:

```
python setup.py build_ext -i
```

2.16.3 Running Tests

It is recommended to use `pytest` to run Mars tests. A simple command below will run all the tests of Mars:

```
pytest mars
```

If you want to generate a coverage report as well, you can run:

```
pytest --cov=mars --cov-report=html mars
```

Coverage report will be put into the directory `htmlcov`.

The command above does not contain coverage data for Cython files by default. To obtain coverage data about Cython files, you can run

```
CYTHON_TRACE=1 python setup.py build_ext -i --force
```

before running the pytest command mentioned above. After report is generated, it it recommended to remove all generated C files and binaries and rebuild without `CYTHON_TRACE`, as this option will reduce the performance of Mars.

2.16.4 Building Documentations

Mars uses `sphinx` to build documents. You need to install necessary packages with the command below to install these dependencies and build your documents into HTML.

```
pip install -r docs/doc-requirements.txt
cd docs
make html
```

The built documents are in `docs/build/html` directory.

When you want to create translations of Mars documents, you may append `-l <locale>` after the `I18NSPHINXLANGS` variable in `Makefile`. Currently only simplified Chinese is supported. After that, run the command below to generate portable files (`*.po`) for the documents, which are in `docs/source/locale/<locale>/LC_MESSAGES`:

```
cd docs
make gettext
```

After that you can translate Mars documents into your language. Note that when you run `make gettext` again, translations will be broken into a fixed-width text. For Chinese translators, you need to install `jieba` to get this effect.

When you finish translation, you can run

```
cd docs
# change LANG into the language you want to build
make -e SPHINXOPTS="-D language='LANG'" html
```

to build the document in the language you just translated into.

Bibliography

[CT] Cooley, James W., and John W. Tukey, 1965, "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.* 19: 297-301.

Python Module Index

m

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

U

V

W

Z