Module: tf.linalg

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg#top_of_page)
* [Classes](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg#classes)
* [Functions](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg#functions)

Operations for linear algebra.

Classes

[class LinearOperator](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperator): Base class defining a [batch of] linear operator[s].

[class LinearOperatorAdjoint](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorAdjoint): LinearOperator representing the adjoint of another operator.

[class LinearOperatorBlockDiag](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorBlockDiag): Combines one or more LinearOperators in to a Block Diagonal matrix.

[class LinearOperatorCirculant](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorCirculant): LinearOperator acting like a circulant matrix.

[class LinearOperatorCirculant2D](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorCirculant2D): LinearOperator acting like a block circulant matrix.

[class LinearOperatorCirculant3D](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorCirculant3D): LinearOperator acting like a nested block circulant matrix.

[class LinearOperatorComposition](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorComposition): Composes one or more LinearOperators.

[class LinearOperatorDiag](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorDiag): LinearOperator acting like a [batch] square diagonal matrix.

[class LinearOperatorFullMatrix](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorFullMatrix): LinearOperator that wraps a [batch] matrix.

[class LinearOperatorHouseholder](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorHouseholder): LinearOperator acting like a [batch] of Householder transformations.

[class LinearOperatorIdentity](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorIdentity): LinearOperator acting like a [batch] square identity matrix.

[class LinearOperatorInversion](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorInversion): LinearOperator representing the inverse of another operator.

[class LinearOperatorKronecker](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorKronecker): Kronecker product between two LinearOperators.

[class LinearOperatorLowRankUpdate](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorLowRankUpdate): Perturb a LinearOperator with a rank K update.

[class LinearOperatorLowerTriangular](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorLowerTriangular): LinearOperator acting like a [batch] square lower triangular matrix.

[class LinearOperatorScaledIdentity](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorScaledIdentity): LinearOperator acting like a scaled [batch] identity matrix A = c I.

[class LinearOperatorToeplitz](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorToeplitz): LinearOperator acting like a [batch] of toeplitz matrices.

[class LinearOperatorZeros](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorZeros): LinearOperator acting like a [batch] zero matrix.

Functions

[adjoint(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/adjoint): Transposes the last two dimensions of and conjugates tensor matrix.

[band\_part(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/band_part): Copy a tensor setting everything outside a central band in each innermost matrix

[cholesky(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/cholesky): Computes the Cholesky decomposition of one or more square matrices.

[cholesky\_solve(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/cholesky_solve): Solves systems of linear eqns A X = RHS, given Cholesky factorizations.

[cross(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/cross): Compute the pairwise cross product.

[det(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/det): Computes the determinant of one or more square matrices.

[diag(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/diag): Returns a batched diagonal tensor with a given batched diagonal values.

[diag\_part(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/diag_part): Returns the batched diagonal part of a batched tensor.

[eigh(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/eigh): Computes the eigen decomposition of a batch of self-adjoint matrices.

[eigvalsh(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/eigvalsh): Computes the eigenvalues of one or more self-adjoint matrices.

[einsum(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/einsum): A generalized contraction between tensors of arbitrary dimension.

[expm(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/expm): Computes the matrix exponential of one or more square matrices.

[eye(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/eye): Construct an identity matrix, or a batch of matrices.

[global\_norm(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/global_norm): Computes the global norm of multiple tensors.

[inv(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/inv): Computes the inverse of one or more square invertible matrices or their

[l2\_normalize(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/math/l2_normalize): Normalizes along dimension axis using an L2 norm.

[logdet(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/logdet): Computes log of the determinant of a hermitian positive definite matrix.

[logm(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/logm): Computes the matrix logarithm of one or more square matrices:

[lstsq(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/lstsq): Solves one or more linear least-squares problems.

[lu(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/lu): Computes the LU decomposition of one or more square matrices.

[matmul(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/matmul): Multiplies matrix a by matrix b, producing a \* b.

[matrix\_transpose(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/matrix_transpose): Transposes last two dimensions of tensor a.

[matvec(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/matvec): Multiplies matrix a by vector b, producing a \* b.

[norm(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/norm): Computes the norm of vectors, matrices, and tensors.

[qr(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/qr): Computes the QR decompositions of one or more matrices.

[set\_diag(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/set_diag): Returns a batched matrix tensor with new batched diagonal values.

[slogdet(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/slogdet): Computes the sign and the log of the absolute value of the determinant of

[solve(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/solve): Solves systems of linear equations.

[sqrtm(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/sqrtm): Computes the matrix square root of one or more square matrices:

[svd(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/svd): Computes the singular value decompositions of one or more matrices.

[tensor\_diag(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/tensor_diag): Returns a diagonal tensor with a given diagonal values.

[tensor\_diag\_part(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/tensor_diag_part): Returns the diagonal part of the tensor.

[tensordot(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/tensordot): Tensor contraction of a and b along specified axes.

[trace(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/trace): Compute the trace of a tensor x.

[triangular\_solve(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/triangular_solve): Solves systems of linear equations with upper or lower triangular matrices by backsubstitution.

[tridiagonal\_matmul(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/tridiagonal_matmul): Multiplies tridiagonal matrix by matrix.

[tridiagonal\_solve(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/tridiagonal_solve): Solves tridiagonal systems of equations.

# tf.compat.v1.linalg.l2\_normalize

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/compat/v1/linalg/l2_normalize#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/compat/v1/linalg/l2_normalize#aliases)

Normalizes along dimension axis using an L2 norm. (deprecated arguments)

### Aliases:

* tf.compat.v1.linalg.l2\_normalize
* tf.compat.v1.math.l2\_normalize
* tf.compat.v1.nn.l2\_normalize

tf.compat.v1.linalg.l2\_normalize(  
    x,  
    axis=None,  
    epsilon=1e-12,  
    name=None,  
    dim=None  
)

Defined in [python/ops/nn\_impl.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/nn_impl.py).

**Warning:** SOME ARGUMENTS ARE DEPRECATED: **(dim)**. They will be removed in a future version. Instructions for updating: dim is deprecated, use axis instead

For a 1-D tensor with axis = 0, computes

output = x / sqrt(max(sum(x\*\*2), epsilon))

For x with more dimensions, independently normalizes each 1-D slice along dimension axis.

#### Args:

* **x**: A Tensor.
* **axis**: Dimension along which to normalize. A scalar or a vector of integers.
* **epsilon**: A lower bound value for the norm. Will use sqrt(epsilon) as the divisor if norm < sqrt(epsilon).
* **name**: A name for this operation (optional).
* **dim**: Deprecated alias for axis.

#### Returns:

A Tensor with the same shape as x.

# tf.linalg.adjoint

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/adjoint#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/adjoint#aliases)

Transposes the last two dimensions of and conjugates tensor matrix.

### Aliases:

* tf.compat.v1.linalg.adjoint
* tf.compat.v2.linalg.adjoint
* tf.linalg.adjoint

tf.linalg.adjoint(  
    matrix,  
    name=None  
)

Defined in [python/ops/linalg/linalg\_impl.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg/linalg_impl.py).

#### For example:

x = tf.constant([[1 + 1j, 2 + 2j, 3 + 3j],  
                 [4 + 4j, 5 + 5j, 6 + 6j]])  
tf.linalg.adjoint(x)  # [[1 - 1j, 4 - 4j],  
                      #  [2 - 2j, 5 - 5j],  
                      #  [3 - 3j, 6 - 6j]]

#### Args:

* **matrix**: A Tensor. Must be float16, float32, float64, complex64, or complex128 with shape [..., M, M].
* **name**: A name to give this Op (optional).

#### Returns:

The adjoint (a.k.a. Hermitian transpose a.k.a. conjugate transpose) of matrix.

# tf.linalg.band\_part

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/band_part#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/band_part#aliases)
* [Used in the tutorials:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/band_part#used_in_the_tutorials)

Copy a tensor setting everything outside a central band in each innermost matrix

### Aliases:

* tf.compat.v1.linalg.band\_part
* tf.compat.v1.matrix\_band\_part
* tf.compat.v2.linalg.band\_part
* tf.linalg.band\_part

tf.linalg.band\_part(  
    input,  
    num\_lower,  
    num\_upper,  
    name=None  
)

Defined in generated file: python/ops/gen\_array\_ops.py.

### Used in the tutorials:

* [Transformer model for language understanding](https://www.tensorflow.org/beta/tutorials/text/transformer)

to zero.

The band part is computed as follows: Assume input has k dimensions [I, J, K, ..., M, N], then the output is a tensor with the same shape where

band[i, j, k, ..., m, n] = in\_band(m, n) \* input[i, j, k, ..., m, n].

The indicator function

in\_band(m, n) = (num\_lower < 0 || (m-n) <= num\_lower)) && (num\_upper < 0 || (n-m) <= num\_upper).

#### For example:

# if 'input' is [[ 0,  1,  2, 3]  
                 [-1,  0,  1, 2]  
                 [-2, -1,  0, 1]  
                 [-3, -2, -1, 0]],  
  
tf.matrix\_band\_part(input, 1, -1) ==> [[ 0,  1,  2, 3]  
                                       [-1,  0,  1, 2]  
                                       [ 0, -1,  0, 1]  
                                       [ 0,  0, -1, 0]],  
  
tf.matrix\_band\_part(input, 2, 1) ==> [[ 0,  1,  0, 0]  
                                      [-1,  0,  1, 0]  
                                      [-2, -1,  0, 1]  
                                      [ 0, -2, -1, 0]]

#### Useful special cases:

 tf.matrix\_band\_part(input, 0, -1) ==> Upper triangular part.  
 tf.matrix\_band\_part(input, -1, 0) ==> Lower triangular part.  
 tf.matrix\_band\_part(input, 0, 0) ==> Diagonal.

#### Args:

* **input**: A Tensor. Rank k tensor.
* **num\_lower**: A Tensor. Must be one of the following types: int32, int64. 0-D tensor. Number of subdiagonals to keep. If negative, keep entire lower triangle.
* **num\_upper**: A Tensor. Must have the same type as num\_lower. 0-D tensor. Number of superdiagonals to keep. If negative, keep entire upper triangle.
* **name**: A name for the operation (optional).

#### Returns:

A Tensor. Has the same type as input.

# tf.linalg.cholesky

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/cholesky#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/cholesky#aliases)

Computes the Cholesky decomposition of one or more square matrices.

### Aliases:

* tf.compat.v1.cholesky
* tf.compat.v1.linalg.cholesky
* tf.compat.v2.linalg.cholesky
* tf.linalg.cholesky

tf.linalg.cholesky(  
    input,  
    name=None  
)

Defined in generated file: python/ops/gen\_linalg\_ops.py.

The input is a tensor of shape [..., M, M] whose inner-most 2 dimensions form square matrices.

The input has to be symmetric and positive definite. Only the lower-triangular part of the input will be used for this operation. The upper-triangular part will not be read.

The output is a tensor of the same shape as the input containing the Cholesky decompositions for all input submatrices [..., :, :].

**Note**: The gradient computation on GPU is faster for large matrices but not for large batch dimensions when the submatrices are small. In this case it might be faster to use the CPU.

#### Args:

* **input**: A Tensor. Must be one of the following types: float64, float32, half, complex64, complex128. Shape is [..., M, M].
* **name**: A name for the operation (optional).

#### Returns:

A Tensor. Has the same type as input.

# tf.linalg.cholesky\_solve

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/cholesky_solve#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/cholesky_solve#aliases)

Solves systems of linear eqns A X = RHS, given Cholesky factorizations.

### Aliases:

* tf.compat.v1.cholesky\_solve
* tf.compat.v1.linalg.cholesky\_solve
* tf.compat.v2.linalg.cholesky\_solve
* tf.linalg.cholesky\_solve

tf.linalg.cholesky\_solve(  
    chol,  
    rhs,  
    name=None  
)

Defined in [python/ops/linalg\_ops.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg_ops.py).

# Solve 10 separate 2x2 linear systems:  
A = ... # shape 10 x 2 x 2  
RHS = ... # shape 10 x 2 x 1  
chol = tf.linalg.cholesky(A)  # shape 10 x 2 x 2  
X = tf.linalg.cholesky\_solve(chol, RHS)  # shape 10 x 2 x 1  
# tf.matmul(A, X) ~ RHS  
X[3, :, 0]  # Solution to the linear system A[3, :, :] x = RHS[3, :, 0]  
  
# Solve five linear systems (K = 5) for every member of the length 10 batch.  
A = ... # shape 10 x 2 x 2  
RHS = ... # shape 10 x 2 x 5  
...  
X[3, :, 2]  # Solution to the linear system A[3, :, :] x = RHS[3, :, 2]

#### Args:

* **chol**: A Tensor. Must be float32 or float64, shape is [..., M, M]. Cholesky factorization of A, e.g. chol = tf.linalg.cholesky(A). For that reason, only the lower triangular parts (including the diagonal) of the last two dimensions of chol are used. The strictly upper part is assumed to be zero and not accessed.
* **rhs**: A Tensor, same type as chol, shape is [..., M, K].
* **name**: A name to give this Op. Defaults to cholesky\_solve.

#### Returns:

Solution to A x = rhs, shape [..., M, K].

# tf.linalg.cross

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/cross#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/cross#aliases)

Compute the pairwise cross product.

### Aliases:

* tf.compat.v1.cross
* tf.compat.v1.linalg.cross
* tf.compat.v2.linalg.cross
* tf.linalg.cross

tf.linalg.cross(  
    a,  
    b,  
    name=None  
)

Defined in generated file: python/ops/gen\_math\_ops.py.

a and b must be the same shape; they can either be simple 3-element vectors, or any shape where the innermost dimension is 3. In the latter case, each pair of corresponding 3-element vectors is cross-multiplied independently.

#### Args:

* **a**: A Tensor. Must be one of the following types: float32, float64, int32, uint8, int16, int8, int64, bfloat16, uint16, half, uint32, uint64. A tensor containing 3-element vectors.
* **b**: A Tensor. Must have the same type as a. Another tensor, of same type and shape as a.
* **name**: A name for the operation (optional).

#### Returns:

A Tensor. Has the same type as a.

# tf.linalg.det

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/det#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/det#aliases)

Computes the determinant of one or more square matrices.

### Aliases:

* tf.compat.v1.linalg.det
* tf.compat.v1.matrix\_determinant
* tf.compat.v2.linalg.det
* tf.linalg.det

tf.linalg.det(  
    input,  
    name=None  
)

Defined in generated file: python/ops/gen\_linalg\_ops.py.

The input is a tensor of shape [..., M, M] whose inner-most 2 dimensions form square matrices. The output is a tensor containing the determinants for all input submatrices [..., :, :].

#### Args:

* **input**: A Tensor. Must be one of the following types: half, float32, float64, complex64, complex128. Shape is [..., M, M].
* **name**: A name for the operation (optional).

#### Returns:

A Tensor. Has the same type as input.

# tf.linalg.diag

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/diag#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/diag#aliases)

Returns a batched diagonal tensor with a given batched diagonal values.

### Aliases:

* tf.compat.v1.linalg.diag
* tf.compat.v1.matrix\_diag
* tf.compat.v2.linalg.diag
* tf.linalg.diag

tf.linalg.diag(  
    diagonal,  
    name=None  
)

Defined in generated file: python/ops/gen\_array\_ops.py.

Given a diagonal, this operation returns a tensor with the diagonal and everything else padded with zeros. The diagonal is computed as follows:

Assume diagonal has k dimensions [I, J, K, ..., N], then the output is a tensor of rank k+1with dimensions [I, J, K, ..., N, N]` where:

output[i, j, k, ..., m, n] = 1{m=n} \* diagonal[i, j, k, ..., n].

#### For example:

# 'diagonal' is [[1, 2, 3, 4], [5, 6, 7, 8]]  
  
and diagonal.shape = (2, 4)  
  
tf.matrix\_diag(diagonal) ==> [[[1, 0, 0, 0]  
                                     [0, 2, 0, 0]  
                                     [0, 0, 3, 0]  
                                     [0, 0, 0, 4]],  
                                    [[5, 0, 0, 0]  
                                     [0, 6, 0, 0]  
                                     [0, 0, 7, 0]  
                                     [0, 0, 0, 8]]]  
  
which has shape (2, 4, 4)

#### Args:

* **diagonal**: A Tensor. Rank k, where k >= 1.
* **name**: A name for the operation (optional).

#### Returns:

A Tensor. Has the same type as diagonal.

# tf.linalg.diag\_part

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/diag_part#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/diag_part#aliases)

Returns the batched diagonal part of a batched tensor.

### Aliases:

* tf.compat.v1.linalg.diag\_part
* tf.compat.v1.matrix\_diag\_part
* tf.compat.v2.linalg.diag\_part
* tf.linalg.diag\_part

tf.linalg.diag\_part(  
    input,  
    name=None  
)

Defined in generated file: python/ops/gen\_array\_ops.py.

This operation returns a tensor with the diagonal part of the batched input. The diagonal part is computed as follows:

Assume input has k dimensions [I, J, K, ..., M, N], then the output is a tensor of rank k - 1 with dimensions [I, J, K, ..., min(M, N)] where:

diagonal[i, j, k, ..., n] = input[i, j, k, ..., n, n].

The input must be at least a matrix.

#### For example:

# 'input' is [[[1, 0, 0, 0]  
               [0, 2, 0, 0]  
               [0, 0, 3, 0]  
               [0, 0, 0, 4]],  
              [[5, 0, 0, 0]  
               [0, 6, 0, 0]  
               [0, 0, 7, 0]  
               [0, 0, 0, 8]]]  
  
and input.shape = (2, 4, 4)  
  
tf.matrix\_diag\_part(input) ==> [[1, 2, 3, 4], [5, 6, 7, 8]]  
  
which has shape (2, 4)

#### Args:

* **input**: A Tensor. Rank k tensor where k >= 2.
* **name**: A name for the operation (optional).

#### Returns:

A Tensor. Has the same type as input.

# tf.linalg.eigh

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/eigh#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/eigh#aliases)

Computes the eigen decomposition of a batch of self-adjoint matrices.

### Aliases:

* tf.compat.v1.linalg.eigh
* tf.compat.v1.self\_adjoint\_eig
* tf.compat.v2.linalg.eigh
* tf.linalg.eigh

tf.linalg.eigh(  
    tensor,  
    name=None  
)

Defined in [python/ops/linalg\_ops.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg_ops.py).

Computes the eigenvalues and eigenvectors of the innermost N-by-N matrices in tensor such thattensor[...,:,:] \* v[..., :,i] = e[..., i] \* v[...,:,i], for i=0...N-1.

#### Args:

* **tensor**: Tensor of shape [..., N, N]. Only the lower triangular part of each inner inner matrix is referenced.
* **name**: string, optional name of the operation.

#### Returns:

* **e**: Eigenvalues. Shape is [..., N]. Sorted in non-decreasing order.
* **v**: Eigenvectors. Shape is [..., N, N]. The columns of the inner most matrices contain eigenvectors of the corresponding matrices in tensor

# tf.linalg.eigvalsh

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/eigvalsh#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/eigvalsh#aliases)

Computes the eigenvalues of one or more self-adjoint matrices.

### Aliases:

* tf.compat.v1.linalg.eigvalsh
* tf.compat.v1.self\_adjoint\_eigvals
* tf.compat.v2.linalg.eigvalsh
* tf.linalg.eigvalsh

tf.linalg.eigvalsh(  
    tensor,  
    name=None  
)

Defined in [python/ops/linalg\_ops.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg_ops.py).

**Note:** If your program backpropagates through this function, you should replace it with a call to tf.linalg.eigh (possibly ignoring the second output) to avoid computing the eigen decomposition twice. This is because the eigenvectors are used to compute the gradient w.r.t. the eigenvalues. See \_SelfAdjointEigV2Grad in linalg\_grad.py.

#### Args:

* **tensor**: Tensor of shape [..., N, N].
* **name**: string, optional name of the operation.

#### Returns:

* **e**: Eigenvalues. Shape is [..., N]. The vector e[..., :] contains the N eigenvalues of tensor[..., :, :].

# tf.linalg.expm

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/expm#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/expm#aliases)

Computes the matrix exponential of one or more square matrices.

### Aliases:

* tf.compat.v1.linalg.expm
* tf.compat.v2.linalg.expm
* tf.linalg.expm

tf.linalg.expm(  
    input,  
    name=None  
)

Defined in [python/ops/linalg/linalg\_impl.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg/linalg_impl.py).

exp(A) = \sum\_{n=0}^\infty A^n/n!

The exponential is computed using a combination of the scaling and squaring method and the Pade approximation. Details can be found in: Nicholas J. Higham, "The scaling and squaring method for the matrix exponential revisited," SIAM J. Matrix Anal. Applic., 26:1179-1193, 2005.

The input is a tensor of shape [..., M, M] whose inner-most 2 dimensions form square matrices. The output is a tensor of the same shape as the input containing the exponential for all input submatrices [..., :, :].

#### Args:

* **input**: A Tensor. Must be float16, float32, float64, complex64, or complex128 with shape [..., M, M].
* **name**: A name to give this Op (optional).

#### Returns:

the matrix exponential of the input.

#### Raises:

* **ValueError**: An unsupported type is provided as input.

#### Scipy Compatibility

Equivalent to scipy.linalg.expm

# tf.linalg.global\_norm

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/global_norm#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/global_norm#aliases)

Computes the global norm of multiple tensors.

### Aliases:

* tf.compat.v1.global\_norm
* tf.compat.v1.linalg.global\_norm
* tf.compat.v2.linalg.global\_norm
* tf.linalg.global\_norm

tf.linalg.global\_norm(  
    t\_list,  
    name=None  
)

Defined in [python/ops/clip\_ops.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/clip_ops.py).

Given a tuple or list of tensors t\_list, this operation returns the global norm of the elements in all tensors in t\_list. The global norm is computed as:

global\_norm = sqrt(sum([l2norm(t)\*\*2 for t in t\_list]))

Any entries in t\_list that are of type None are ignored.

#### Args:

* **t\_list**: A tuple or list of mixed Tensors, IndexedSlices, or None.
* **name**: A name for the operation (optional).

#### Returns:

A 0-D (scalar) Tensor of type float.

#### Raises:

* **TypeError**: If t\_list is not a sequence.

# tf.linalg.inv

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/inv#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/inv#aliases)

Computes the inverse of one or more square invertible matrices or their

### Aliases:

* tf.compat.v1.linalg.inv
* tf.compat.v1.matrix\_inverse
* tf.compat.v2.linalg.inv
* tf.linalg.inv

tf.linalg.inv(  
    input,  
    adjoint=False,  
    name=None  
)

Defined in generated file: python/ops/gen\_linalg\_ops.py.

adjoints (conjugate transposes).

The input is a tensor of shape [..., M, M] whose inner-most 2 dimensions form square matrices. The output is a tensor of the same shape as the input containing the inverse for all input submatrices [..., :, :].

The op uses LU decomposition with partial pivoting to compute the inverses.

If a matrix is not invertible there is no guarantee what the op does. It may detect the condition and raise an exception or it may simply return a garbage result.

#### Args:

* **input**: A Tensor. Must be one of the following types: float64, float32, half, complex64, complex128. Shape is [..., M, M].
* **adjoint**: An optional bool. Defaults to False.
* **name**: A name for the operation (optional).

#### Returns:

A Tensor. Has the same type as input.

# tf.linalg.LinearOperator

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperator#top_of_page)
* [Class LinearOperator](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperator#class_linearoperator)
  + [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperator#aliases)
* [\_\_init\_\_](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperator#__init__)
* [Properties](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperator#properties)

## Class LinearOperator

Base class defining a [batch of] linear operator[s].

### Aliases:

* Class tf.compat.v1.linalg.LinearOperator
* Class tf.compat.v2.linalg.LinearOperator
* Class tf.linalg.LinearOperator

Defined in [python/ops/linalg/linear\_operator.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg/linear_operator.py).

Subclasses of LinearOperator provide access to common methods on a (batch) matrix, without the need to materialize the matrix. This allows:

* Matrix free computations
* Operators that take advantage of special structure, while providing a consistent API to users.

#### Subclassing

To enable a public method, subclasses should implement the leading-underscore version of the method. The argument signature should be identical except for the omission of name="...". For example, to enable matmul(x, adjoint=False, name="matmul") a subclass should implement\_matmul(x, adjoint=False).

#### Performance contract

Subclasses should only implement the assert methods (e.g. assert\_non\_singular) if they can be done in less than O(N^3) time.

Class docstrings should contain an explanation of computational complexity. Since this is a high-performance library, attention should be paid to detail, and explanations can include constants as well as Big-O notation.

#### Shape compatibility

LinearOperator subclasses should operate on a [batch] matrix with compatible shape. Class docstrings should define what is meant by compatible shape. Some subclasses may not support batching.

#### Examples:

x is a batch matrix with compatible shape for matmul if

operator.shape = [B1,...,Bb] + [M, N],  b >= 0,  
x.shape =   [B1,...,Bb] + [N, R]

rhs is a batch matrix with compatible shape for solve if

operator.shape = [B1,...,Bb] + [M, N],  b >= 0,  
rhs.shape =   [B1,...,Bb] + [M, R]

#### Example docstring for subclasses.

This operator acts like a (batch) matrix A with shape [B1,...,Bb, M, N] for some b >= 0. The first b indices index a batch member. For every batch index (i1,...,ib), A[i1,...,ib, : :] is an m x n matrix. Again, this matrix A may not be materialized, but for purposes of identifying and working with compatible arguments the shape is relevant.

#### Examples:

some\_tensor = ... shape = ????  
operator = MyLinOp(some\_tensor)  
  
operator.shape()  
==> [2, 4, 4]  
  
operator.log\_abs\_determinant()  
==> Shape [2] Tensor  
  
x = ... Shape [2, 4, 5] Tensor  
  
operator.matmul(x)  
==> Shape [2, 4, 5] Tensor

#### Shape compatibility

This operator acts on batch matrices with compatible shape. FILL IN WHAT IS MEANT BY COMPATIBLE SHAPE

#### Performance

FILL THIS IN

#### Matrix property hints

This LinearOperator is initialized with boolean flags of the form is\_X, for X = non\_singular, self\_adjoint, positive\_definite, square. These have the following meaning:

* If is\_X == True, callers should expect the operator to have the property X. This is a promise that should be fulfilled, but is not a runtime assert. For example, finite floating point precision may result in these promises being violated.
* If is\_X == False, callers should expect the operator to not have X.
* If is\_X == None (the default), callers should have no expectation either way.

## \_\_init\_\_

\_\_init\_\_(  
    dtype,  
    graph\_parents=None,  
    is\_non\_singular=None,  
    is\_self\_adjoint=None,  
    is\_positive\_definite=None,  
    is\_square=None,  
    name=None  
)

Initialize the LinearOperator.

**This is a private method for subclass use.** **Subclasses should copy-paste this \_\_init\_\_documentation.**

#### Args:

* **dtype**: The type of the this LinearOperator. Arguments to matmul and solve will have to be this type.
* **graph\_parents**: Python list of graph prerequisites of this LinearOperator Typically tensors that are passed during initialization.
* **is\_non\_singular**: Expect that this operator is non-singular.
* **is\_self\_adjoint**: Expect that this operator is equal to its hermitian transpose. If dtype is real, this is equivalent to being symmetric.
* **is\_positive\_definite**: Expect that this operator is positive definite, meaning the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive-definite. See: https://en.wikipedia.org/wiki/Positive-definite\_matrix#Extension\_for\_non-symmetric\_matrices
* **is\_square**: Expect that this operator acts like square [batch] matrices.
* **name**: A name for this LinearOperator.

#### Raises:

* **ValueError**: If any member of graph\_parents is None or not a Tensor.
* **ValueError**: If hints are set incorrectly.

## Properties

### H

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### batch\_shape

TensorShape of batch dimensions of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb]), equivalent to A.get\_shape()[:-2]

#### Returns:

TensorShape, statically determined, may be undefined.

### domain\_dimension

Dimension (in the sense of vector spaces) of the domain of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Returns:

Dimension object.

### dtype

The DType of Tensors handled by this LinearOperator.

### graph\_parents

List of graph dependencies of this LinearOperator.

### is\_non\_singular

### is\_positive\_definite

### is\_self\_adjoint

### is\_square

Return True/False depending on if this operator is square.

### name

Name prepended to all ops created by this LinearOperator.

### range\_dimension

Dimension (in the sense of vector spaces) of the range of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Returns:

Dimension object.

### shape

TensorShape of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb, M, N]), equivalent to A.get\_shape().

#### Returns:

TensorShape, statically determined, may be undefined.

### tensor\_rank

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

Python integer, or None if the tensor rank is undefined.

## Methods

### add\_to\_tensor

add\_to\_tensor(  
    x,  
    name='add\_to\_tensor'  
)

Add matrix represented by this operator to x. Equivalent to A + x.

#### Args:

* **x**: Tensor with same dtype and shape broadcastable to self.shape.
* **name**: A name to give this Op.

#### Returns:

A Tensor with broadcast shape and same dtype as self.

### adjoint

adjoint(name='adjoint')

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### assert\_non\_singular

assert\_non\_singular(name='assert\_non\_singular')

Returns an Op that asserts this operator is non singular.

This operator is considered non-singular if

ConditionNumber < max{100, range\_dimension, domain\_dimension} \* eps,  
eps := np.finfo(self.dtype.as\_numpy\_dtype).eps

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is singular.

### assert\_positive\_definite

assert\_positive\_definite(name='assert\_positive\_definite')

Returns an Op that asserts this operator is positive definite.

Here, positive definite means that the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive definite.

#### Args:

* **name**: A name to give this Op.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not positive definite.

### assert\_self\_adjoint

assert\_self\_adjoint(name='assert\_self\_adjoint')

Returns an Op that asserts this operator is self-adjoint.

Here we check that this operator is exactly equal to its hermitian transpose.

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not self-adjoint.

### batch\_shape\_tensor

batch\_shape\_tensor(name='batch\_shape\_tensor')

Shape of batch dimensions of this operator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb].

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### cholesky

cholesky(name='cholesky')

Returns a Cholesky factor as a LinearOperator.

Given A representing this LinearOperator, if A is positive definite self-adjoint, return L, where A = L L^T, i.e. the cholesky decomposition.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the lower triangular matrix in the Cholesky decomposition.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be positive definite and self adjoint.

### determinant

determinant(name='det')

Determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### diag\_part

diag\_part(name='diag\_part')

Efficiently get the [batch] diagonal part of this operator.

If this operator has shape [B1,...,Bb, M, N], this returns a Tensor diagonal, of shape [B1,...,Bb, min(M, N)], where diagonal[b1,...,bb, i] = self.to\_dense()[b1,...,bb, i, i].

my\_operator = LinearOperatorDiag([1., 2.])  
  
# Efficiently get the diagonal  
my\_operator.diag\_part()  
==> [1., 2.]  
  
# Equivalent, but inefficient method  
tf.linalg.diag\_part(my\_operator.to\_dense())  
==> [1., 2.]

#### Args:

* **name**: A name for this Op.

#### Returns:

* **diag\_part**: A Tensor of same dtype as self.

### domain\_dimension\_tensor

domain\_dimension\_tensor(name='domain\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the domain of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### inverse

inverse(name='inverse')

Returns the Inverse of this LinearOperator.

Given A representing this LinearOperator, return a LinearOperator representing A^-1.

#### Args:

* **name**: A name scope to use for ops added by this method.

#### Returns:

LinearOperator representing inverse of this matrix.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be non\_singular.

### log\_abs\_determinant

log\_abs\_determinant(name='log\_abs\_det')

Log absolute value of determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### matmul

matmul(  
    x,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='matmul'  
)

Transform [batch] matrix x with left multiplication: x --> Ax.

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
X = ... # shape [..., N, R], batch matrix, R > 0.  
  
Y = operator.matmul(X)  
Y.shape  
==> [..., M, R]  
  
Y[..., :, r] = sum\_j A[..., :, j] X[j, r]

#### Args:

* **x**: LinearOperator or Tensor with compatible shape and same dtype as self. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **adjoint\_arg**: Python bool. If True, compute A x^H where x^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name for this Op.

#### Returns:

A LinearOperator or Tensor with shape [..., M, R] and same dtype as self.

### matvec

matvec(  
    x,  
    adjoint=False,  
    name='matvec'  
)

Transform [batch] vector x with left multiplication: x --> Ax.

# Make an operator acting like batch matric A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
  
X = ... # shape [..., N], batch vector  
  
Y = operator.matvec(X)  
Y.shape  
==> [..., M]  
  
Y[..., :] = sum\_j A[..., :, j] X[..., j]

#### Args:

* **x**: Tensor with compatible shape and same dtype as self. x is treated as a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **name**: A name for this Op.

#### Returns:

A Tensor with shape [..., M] and same dtype as self.

### range\_dimension\_tensor

range\_dimension\_tensor(name='range\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the range of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### shape\_tensor

shape\_tensor(name='shape\_tensor')

Shape of this LinearOperator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb, M, N], equivalent to tf.shape(A).

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### solve

solve(  
    rhs,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='solve'  
)

Solve (exact or approx) R (batch) systems of equations: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve R > 0 linear systems for every member of the batch.  
RHS = ... # shape [..., M, R]  
  
X = operator.solve(RHS)  
# X[..., :, r] is the solution to the r'th linear system  
# sum\_j A[..., :, j] X[..., j, r] = RHS[..., :, r]  
  
operator.matmul(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator and compatible shape. rhs is treated like a [batch] matrix meaning for every set of leading dimensions, the last two dimensions defines a matrix. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **adjoint\_arg**: Python bool. If True, solve A X = rhs^H where rhs^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N, R] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### solvevec

solvevec(  
    rhs,  
    adjoint=False,  
    name='solve'  
)

Solve single equation with best effort: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve one linear system for every member of the batch.  
RHS = ... # shape [..., M]  
  
X = operator.solvevec(RHS)  
# X is the solution to the linear system  
# sum\_j A[..., :, j] X[..., j] = RHS[..., :]  
  
operator.matvec(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator. rhs is treated like a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility regarding batch dimensions.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### tensor\_rank\_tensor

tensor\_rank\_tensor(name='tensor\_rank\_tensor')

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor, determined at runtime.

### to\_dense

to\_dense(name='to\_dense')

Return a dense (batch) matrix representing this operator.

### trace

trace(name='trace')

Trace of the linear operator, equal to sum of self.diag\_part().

If the operator is square, this is also the sum of the eigenvalues.

#### Args:

* **name**: A name for this Op.

#### Returns:

Shape [B1,...,Bb] Tensor of same dtype as self.

# tf.linalg.LinearOperatorAdjoint

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorAdjoint#top_of_page)
* [Class LinearOperatorAdjoint](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorAdjoint#class_linearoperatoradjoint)
  + [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorAdjoint#aliases)
* [\_\_init\_\_](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorAdjoint#__init__)
* [Properties](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorAdjoint#properties)

## Class LinearOperatorAdjoint

LinearOperator representing the adjoint of another operator.

Inherits From: [LinearOperator](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperator)

### Aliases:

* Class tf.compat.v1.linalg.LinearOperatorAdjoint
* Class tf.compat.v2.linalg.LinearOperatorAdjoint
* Class tf.linalg.LinearOperatorAdjoint

Defined in [python/ops/linalg/linear\_operator\_adjoint.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg/linear_operator_adjoint.py).

This operator represents the adjoint of another operator.

# Create a 2 x 2 linear operator.  
operator = LinearOperatorFullMatrix([[1 - i., 3.], [0., 1. + i]])  
operator\_adjoint = LinearOperatorAdjoint(operator)  
  
operator\_adjoint.to\_dense()  
==> [[1. + i, 0.]  
     [3., 1 - i]]  
  
operator\_adjoint.shape  
==> [2, 2]  
  
operator\_adjoint.log\_abs\_determinant()  
==> - log(2)  
  
x = ... Shape [2, 4] Tensor  
operator\_adjoint.matmul(x)  
==> Shape [2, 4] Tensor, equal to operator.matmul(x, adjoint=True)

#### Performance

The performance of LinearOperatorAdjoint depends on the underlying operators performance.

#### Matrix property hints

This LinearOperator is initialized with boolean flags of the form is\_X, for X = non\_singular, self\_adjoint, positive\_definite, square. These have the following meaning:

* If is\_X == True, callers should expect the operator to have the property X. This is a promise that should be fulfilled, but is not a runtime assert. For example, finite floating point precision may result in these promises being violated.
* If is\_X == False, callers should expect the operator to not have X.
* If is\_X == None (the default), callers should have no expectation either way.

## \_\_init\_\_

\_\_init\_\_(  
    operator,  
    is\_non\_singular=None,  
    is\_self\_adjoint=None,  
    is\_positive\_definite=None,  
    is\_square=None,  
    name=None  
)

Initialize a LinearOperatorAdjoint.

LinearOperatorAdjoint is initialized with an operator A. The solve and matmul methods effectively flip the adjoint argument. E.g.

A = MyLinearOperator(...)  
B = LinearOperatorAdjoint(A)  
x = [....]  # a vector  
  
assert A.matvec(x, adjoint=True) == B.matvec(x, adjoint=False)

#### Args:

* **operator**: LinearOperator object.
* **is\_non\_singular**: Expect that this operator is non-singular.
* **is\_self\_adjoint**: Expect that this operator is equal to its hermitian transpose.
* **is\_positive\_definite**: Expect that this operator is positive definite, meaning the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive-definite. See: https://en.wikipedia.org/wiki/Positive-definite\_matrix#Extension\_for\_non-symmetric\_matrices
* **is\_square**: Expect that this operator acts like square [batch] matrices.
* **name**: A name for this LinearOperator. Default is operator.name + "\_adjoint".

#### Raises:

* **ValueError**: If operator.is\_non\_singular is False.

## Properties

### H

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### batch\_shape

TensorShape of batch dimensions of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb]), equivalent to A.get\_shape()[:-2]

#### Returns:

TensorShape, statically determined, may be undefined.

### domain\_dimension

Dimension (in the sense of vector spaces) of the domain of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Returns:

Dimension object.

### dtype

The DType of Tensors handled by this LinearOperator.

### graph\_parents

List of graph dependencies of this LinearOperator.

### is\_non\_singular

### is\_positive\_definite

### is\_self\_adjoint

### is\_square

Return True/False depending on if this operator is square.

### name

Name prepended to all ops created by this LinearOperator.

### operator

The operator before taking the adjoint.

### range\_dimension

Dimension (in the sense of vector spaces) of the range of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Returns:

Dimension object.

### shape

TensorShape of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb, M, N]), equivalent to A.get\_shape().

#### Returns:

TensorShape, statically determined, may be undefined.

### tensor\_rank

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

Python integer, or None if the tensor rank is undefined.

## Methods

### add\_to\_tensor

add\_to\_tensor(  
    x,  
    name='add\_to\_tensor'  
)

Add matrix represented by this operator to x. Equivalent to A + x.

#### Args:

* **x**: Tensor with same dtype and shape broadcastable to self.shape.
* **name**: A name to give this Op.

#### Returns:

A Tensor with broadcast shape and same dtype as self.

### adjoint

adjoint(name='adjoint')

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### assert\_non\_singular

assert\_non\_singular(name='assert\_non\_singular')

Returns an Op that asserts this operator is non singular.

This operator is considered non-singular if

ConditionNumber < max{100, range\_dimension, domain\_dimension} \* eps,  
eps := np.finfo(self.dtype.as\_numpy\_dtype).eps

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is singular.

### assert\_positive\_definite

assert\_positive\_definite(name='assert\_positive\_definite')

Returns an Op that asserts this operator is positive definite.

Here, positive definite means that the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive definite.

#### Args:

* **name**: A name to give this Op.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not positive definite.

### assert\_self\_adjoint

assert\_self\_adjoint(name='assert\_self\_adjoint')

Returns an Op that asserts this operator is self-adjoint.

Here we check that this operator is exactly equal to its hermitian transpose.

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not self-adjoint.

### batch\_shape\_tensor

batch\_shape\_tensor(name='batch\_shape\_tensor')

Shape of batch dimensions of this operator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb].

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### cholesky

cholesky(name='cholesky')

Returns a Cholesky factor as a LinearOperator.

Given A representing this LinearOperator, if A is positive definite self-adjoint, return L, where A = L L^T, i.e. the cholesky decomposition.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the lower triangular matrix in the Cholesky decomposition.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be positive definite and self adjoint.

### determinant

determinant(name='det')

Determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### diag\_part

diag\_part(name='diag\_part')

Efficiently get the [batch] diagonal part of this operator.

If this operator has shape [B1,...,Bb, M, N], this returns a Tensor diagonal, of shape [B1,...,Bb, min(M, N)], where diagonal[b1,...,bb, i] = self.to\_dense()[b1,...,bb, i, i].

my\_operator = LinearOperatorDiag([1., 2.])  
  
# Efficiently get the diagonal  
my\_operator.diag\_part()  
==> [1., 2.]  
  
# Equivalent, but inefficient method  
tf.linalg.diag\_part(my\_operator.to\_dense())  
==> [1., 2.]

#### Args:

* **name**: A name for this Op.

#### Returns:

* **diag\_part**: A Tensor of same dtype as self.

### domain\_dimension\_tensor

domain\_dimension\_tensor(name='domain\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the domain of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### inverse

inverse(name='inverse')

Returns the Inverse of this LinearOperator.

Given A representing this LinearOperator, return a LinearOperator representing A^-1.

#### Args:

* **name**: A name scope to use for ops added by this method.

#### Returns:

LinearOperator representing inverse of this matrix.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be non\_singular.

### log\_abs\_determinant

log\_abs\_determinant(name='log\_abs\_det')

Log absolute value of determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### matmul

matmul(  
    x,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='matmul'  
)

Transform [batch] matrix x with left multiplication: x --> Ax.

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
X = ... # shape [..., N, R], batch matrix, R > 0.  
  
Y = operator.matmul(X)  
Y.shape  
==> [..., M, R]  
  
Y[..., :, r] = sum\_j A[..., :, j] X[j, r]

#### Args:

* **x**: LinearOperator or Tensor with compatible shape and same dtype as self. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **adjoint\_arg**: Python bool. If True, compute A x^H where x^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name for this Op.

#### Returns:

A LinearOperator or Tensor with shape [..., M, R] and same dtype as self.

### matvec

matvec(  
    x,  
    adjoint=False,  
    name='matvec'  
)

Transform [batch] vector x with left multiplication: x --> Ax.

# Make an operator acting like batch matric A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
  
X = ... # shape [..., N], batch vector  
  
Y = operator.matvec(X)  
Y.shape  
==> [..., M]  
  
Y[..., :] = sum\_j A[..., :, j] X[..., j]

#### Args:

* **x**: Tensor with compatible shape and same dtype as self. x is treated as a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **name**: A name for this Op.

#### Returns:

A Tensor with shape [..., M] and same dtype as self.

### range\_dimension\_tensor

range\_dimension\_tensor(name='range\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the range of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### shape\_tensor

shape\_tensor(name='shape\_tensor')

Shape of this LinearOperator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb, M, N], equivalent to tf.shape(A).

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### solve

solve(  
    rhs,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='solve'  
)

Solve (exact or approx) R (batch) systems of equations: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve R > 0 linear systems for every member of the batch.  
RHS = ... # shape [..., M, R]  
  
X = operator.solve(RHS)  
# X[..., :, r] is the solution to the r'th linear system  
# sum\_j A[..., :, j] X[..., j, r] = RHS[..., :, r]  
  
operator.matmul(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator and compatible shape. rhs is treated like a [batch] matrix meaning for every set of leading dimensions, the last two dimensions defines a matrix. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **adjoint\_arg**: Python bool. If True, solve A X = rhs^H where rhs^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N, R] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### solvevec

solvevec(  
    rhs,  
    adjoint=False,  
    name='solve'  
)

Solve single equation with best effort: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve one linear system for every member of the batch.  
RHS = ... # shape [..., M]  
  
X = operator.solvevec(RHS)  
# X is the solution to the linear system  
# sum\_j A[..., :, j] X[..., j] = RHS[..., :]  
  
operator.matvec(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator. rhs is treated like a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility regarding batch dimensions.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### tensor\_rank\_tensor

tensor\_rank\_tensor(name='tensor\_rank\_tensor')

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor, determined at runtime.

### to\_dense

to\_dense(name='to\_dense')

Return a dense (batch) matrix representing this operator.

### trace

trace(name='trace')

Trace of the linear operator, equal to sum of self.diag\_part().

If the operator is square, this is also the sum of the eigenvalues.

#### Args:

* **name**: A name for this Op.

#### Returns:

Shape [B1,...,Bb] Tensor of same dtype as self.

# tf.linalg.LinearOperatorBlockDiag

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorBlockDiag#top_of_page)
* [Class LinearOperatorBlockDiag](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorBlockDiag#class_linearoperatorblockdiag)
  + [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorBlockDiag#aliases)
* [\_\_init\_\_](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorBlockDiag#__init__)
* [Properties](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorBlockDiag#properties)

## Class LinearOperatorBlockDiag

Combines one or more LinearOperators in to a Block Diagonal matrix.

Inherits From: [LinearOperator](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperator)

### Aliases:

* Class tf.compat.v1.linalg.LinearOperatorBlockDiag
* Class tf.compat.v2.linalg.LinearOperatorBlockDiag
* Class tf.linalg.LinearOperatorBlockDiag

Defined in [python/ops/linalg/linear\_operator\_block\_diag.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg/linear_operator_block_diag.py).

This operator combines one or more linear operators [op1,...,opJ], building a new LinearOperator, whose underlying matrix representation is square and has each operator opi on the main diagonal, and zero's elsewhere.

#### Shape compatibility

If opj acts like a [batch] square matrix Aj, then op\_combined acts like the [batch] square matrix formed by having each matrix Aj on the main diagonal.

Each opj is required to represent a square matrix, and hence will have shape batch\_shape\_j + [M\_j, M\_j].

If opj has shape batch\_shape\_j + [M\_j, M\_j], then the combined operator has shape broadcast\_batch\_shape + [sum M\_j, sum M\_j], where broadcast\_batch\_shape is the mutual broadcast of batch\_shape\_j, j = 1,...,J, assuming the intermediate batch shapes broadcast. Even if the combined shape is well defined, the combined operator's methods may fail due to lack of broadcasting ability in the defining operators' methods.

# Create a 4 x 4 linear operator combined of two 2 x 2 operators.  
operator\_1 = LinearOperatorFullMatrix([[1., 2.], [3., 4.]])  
operator\_2 = LinearOperatorFullMatrix([[1., 0.], [0., 1.]])  
operator = LinearOperatorBlockDiag([operator\_1, operator\_2])  
  
operator.to\_dense()  
==> [[1., 2., 0., 0.],  
     [3., 4., 0., 0.],  
     [0., 0., 1., 0.],  
     [0., 0., 0., 1.]]  
  
operator.shape  
==> [4, 4]  
  
operator.log\_abs\_determinant()  
==> scalar Tensor  
  
x1 = ... # Shape [2, 2] Tensor  
x2 = ... # Shape [2, 2] Tensor  
x = tf.concat([x1, x2], 0)  # Shape [2, 4] Tensor  
operator.matmul(x)  
==> tf.concat([operator\_1.matmul(x1), operator\_2.matmul(x2)])  
  
# Create a [2, 3] batch of 4 x 4 linear operators.  
matrix\_44 = tf.random.normal(shape=[2, 3, 4, 4])  
operator\_44 = LinearOperatorFullMatrix(matrix)  
  
# Create a [1, 3] batch of 5 x 5 linear operators.  
matrix\_55 = tf.random.normal(shape=[1, 3, 5, 5])  
operator\_55 = LinearOperatorFullMatrix(matrix\_55)  
  
# Combine to create a [2, 3] batch of 9 x 9 operators.  
operator\_99 = LinearOperatorBlockDiag([operator\_44, operator\_55])  
  
# Create a shape [2, 3, 9] vector.  
x = tf.random.normal(shape=[2, 3, 9])  
operator\_99.matmul(x)  
==> Shape [2, 3, 9] Tensor

#### Performance

The performance of LinearOperatorBlockDiag on any operation is equal to the sum of the individual operators' operations.

#### Matrix property hints

This LinearOperator is initialized with boolean flags of the form is\_X, for X = non\_singular, self\_adjoint, positive\_definite, square. These have the following meaning:

* If is\_X == True, callers should expect the operator to have the property X. This is a promise that should be fulfilled, but is not a runtime assert. For example, finite floating point precision may result in these promises being violated.
* If is\_X == False, callers should expect the operator to not have X.
* If is\_X == None (the default), callers should have no expectation either way.

## \_\_init\_\_

\_\_init\_\_(  
    operators,  
    is\_non\_singular=None,  
    is\_self\_adjoint=None,  
    is\_positive\_definite=None,  
    is\_square=True,  
    name=None  
)

Initialize a LinearOperatorBlockDiag.

LinearOperatorBlockDiag is initialized with a list of operators [op\_1,...,op\_J].

#### Args:

* **operators**: Iterable of LinearOperator objects, each with the same dtype and composable shape.
* **is\_non\_singular**: Expect that this operator is non-singular.
* **is\_self\_adjoint**: Expect that this operator is equal to its hermitian transpose.
* **is\_positive\_definite**: Expect that this operator is positive definite, meaning the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive-definite. See: https://en.wikipedia.org/wiki/Positive-definite\_matrix#Extension\_for\_non-symmetric\_matrices
* **is\_square**: Expect that this operator acts like square [batch] matrices. This is true by default, and will raise a ValueError otherwise.
* **name**: A name for this LinearOperator. Default is the individual operators names joined with \_o\_.

#### Raises:

* **TypeError**: If all operators do not have the same dtype.
* **ValueError**: If operators is empty or are non-square.

## Properties

### H

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### batch\_shape

TensorShape of batch dimensions of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb]), equivalent to A.get\_shape()[:-2]

#### Returns:

TensorShape, statically determined, may be undefined.

### domain\_dimension

Dimension (in the sense of vector spaces) of the domain of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Returns:

Dimension object.

### dtype

The DType of Tensors handled by this LinearOperator.

### graph\_parents

List of graph dependencies of this LinearOperator.

### is\_non\_singular

### is\_positive\_definite

### is\_self\_adjoint

### is\_square

Return True/False depending on if this operator is square.

### name

Name prepended to all ops created by this LinearOperator.

### operators

### range\_dimension

Dimension (in the sense of vector spaces) of the range of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Returns:

Dimension object.

### shape

TensorShape of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb, M, N]), equivalent to A.get\_shape().

#### Returns:

TensorShape, statically determined, may be undefined.

### tensor\_rank

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

Python integer, or None if the tensor rank is undefined.

## Methods

### add\_to\_tensor

add\_to\_tensor(  
    x,  
    name='add\_to\_tensor'  
)

Add matrix represented by this operator to x. Equivalent to A + x.

#### Args:

* **x**: Tensor with same dtype and shape broadcastable to self.shape.
* **name**: A name to give this Op.

#### Returns:

A Tensor with broadcast shape and same dtype as self.

### adjoint

adjoint(name='adjoint')

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### assert\_non\_singular

assert\_non\_singular(name='assert\_non\_singular')

Returns an Op that asserts this operator is non singular.

This operator is considered non-singular if

ConditionNumber < max{100, range\_dimension, domain\_dimension} \* eps,  
eps := np.finfo(self.dtype.as\_numpy\_dtype).eps

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is singular.

### assert\_positive\_definite

assert\_positive\_definite(name='assert\_positive\_definite')

Returns an Op that asserts this operator is positive definite.

Here, positive definite means that the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive definite.

#### Args:

* **name**: A name to give this Op.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not positive definite.

### assert\_self\_adjoint

assert\_self\_adjoint(name='assert\_self\_adjoint')

Returns an Op that asserts this operator is self-adjoint.

Here we check that this operator is exactly equal to its hermitian transpose.

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not self-adjoint.

### batch\_shape\_tensor

batch\_shape\_tensor(name='batch\_shape\_tensor')

Shape of batch dimensions of this operator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb].

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### cholesky

cholesky(name='cholesky')

Returns a Cholesky factor as a LinearOperator.

Given A representing this LinearOperator, if A is positive definite self-adjoint, return L, where A = L L^T, i.e. the cholesky decomposition.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the lower triangular matrix in the Cholesky decomposition.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be positive definite and self adjoint.

### determinant

determinant(name='det')

Determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### diag\_part

diag\_part(name='diag\_part')

Efficiently get the [batch] diagonal part of this operator.

If this operator has shape [B1,...,Bb, M, N], this returns a Tensor diagonal, of shape [B1,...,Bb, min(M, N)], where diagonal[b1,...,bb, i] = self.to\_dense()[b1,...,bb, i, i].

my\_operator = LinearOperatorDiag([1., 2.])  
  
# Efficiently get the diagonal  
my\_operator.diag\_part()  
==> [1., 2.]  
  
# Equivalent, but inefficient method  
tf.linalg.diag\_part(my\_operator.to\_dense())  
==> [1., 2.]

#### Args:

* **name**: A name for this Op.

#### Returns:

* **diag\_part**: A Tensor of same dtype as self.

### domain\_dimension\_tensor

domain\_dimension\_tensor(name='domain\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the domain of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### inverse

inverse(name='inverse')

Returns the Inverse of this LinearOperator.

Given A representing this LinearOperator, return a LinearOperator representing A^-1.

#### Args:

* **name**: A name scope to use for ops added by this method.

#### Returns:

LinearOperator representing inverse of this matrix.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be non\_singular.

### log\_abs\_determinant

log\_abs\_determinant(name='log\_abs\_det')

Log absolute value of determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### matmul

matmul(  
    x,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='matmul'  
)

Transform [batch] matrix x with left multiplication: x --> Ax.

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
X = ... # shape [..., N, R], batch matrix, R > 0.  
  
Y = operator.matmul(X)  
Y.shape  
==> [..., M, R]  
  
Y[..., :, r] = sum\_j A[..., :, j] X[j, r]

#### Args:

* **x**: LinearOperator or Tensor with compatible shape and same dtype as self. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **adjoint\_arg**: Python bool. If True, compute A x^H where x^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name for this Op.

#### Returns:

A LinearOperator or Tensor with shape [..., M, R] and same dtype as self.

### matvec

matvec(  
    x,  
    adjoint=False,  
    name='matvec'  
)

Transform [batch] vector x with left multiplication: x --> Ax.

# Make an operator acting like batch matric A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
  
X = ... # shape [..., N], batch vector  
  
Y = operator.matvec(X)  
Y.shape  
==> [..., M]  
  
Y[..., :] = sum\_j A[..., :, j] X[..., j]

#### Args:

* **x**: Tensor with compatible shape and same dtype as self. x is treated as a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **name**: A name for this Op.

#### Returns:

A Tensor with shape [..., M] and same dtype as self.

### range\_dimension\_tensor

range\_dimension\_tensor(name='range\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the range of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### shape\_tensor

shape\_tensor(name='shape\_tensor')

Shape of this LinearOperator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb, M, N], equivalent to tf.shape(A).

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### solve

solve(  
    rhs,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='solve'  
)

Solve (exact or approx) R (batch) systems of equations: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve R > 0 linear systems for every member of the batch.  
RHS = ... # shape [..., M, R]  
  
X = operator.solve(RHS)  
# X[..., :, r] is the solution to the r'th linear system  
# sum\_j A[..., :, j] X[..., j, r] = RHS[..., :, r]  
  
operator.matmul(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator and compatible shape. rhs is treated like a [batch] matrix meaning for every set of leading dimensions, the last two dimensions defines a matrix. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **adjoint\_arg**: Python bool. If True, solve A X = rhs^H where rhs^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N, R] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### solvevec

solvevec(  
    rhs,  
    adjoint=False,  
    name='solve'  
)

Solve single equation with best effort: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve one linear system for every member of the batch.  
RHS = ... # shape [..., M]  
  
X = operator.solvevec(RHS)  
# X is the solution to the linear system  
# sum\_j A[..., :, j] X[..., j] = RHS[..., :]  
  
operator.matvec(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator. rhs is treated like a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility regarding batch dimensions.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### tensor\_rank\_tensor

tensor\_rank\_tensor(name='tensor\_rank\_tensor')

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor, determined at runtime.

### to\_dense

to\_dense(name='to\_dense')

Return a dense (batch) matrix representing this operator.

### trace

trace(name='trace')

Trace of the linear operator, equal to sum of self.diag\_part().

If the operator is square, this is also the sum of the eigenvalues.

#### Args:

* **name**: A name for this Op.

#### Returns:

Shape [B1,...,Bb] Tensor of same dtype as self.

# tf.linalg.LinearOperatorCirculant

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorCirculant#top_of_page)
* [Class LinearOperatorCirculant](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorCirculant#class_linearoperatorcirculant)
  + [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorCirculant#aliases)
* [\_\_init\_\_](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorCirculant#__init__)
* [Properties](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorCirculant#properties)

## Class LinearOperatorCirculant

LinearOperator acting like a circulant matrix.

### Aliases:

* Class tf.compat.v1.linalg.LinearOperatorCirculant
* Class tf.compat.v2.linalg.LinearOperatorCirculant
* Class tf.linalg.LinearOperatorCirculant

Defined in [python/ops/linalg/linear\_operator\_circulant.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg/linear_operator_circulant.py).

This operator acts like a circulant matrix A with shape [B1,...,Bb, N, N] for some b >= 0. The first b indices index a batch member. For every batch index (i1,...,ib), A[i1,...,ib, : :] is an N x N matrix. This matrix A is not materialized, but for purposes of broadcasting this shape will be relevant.

#### Description in terms of circulant matrices

Circulant means the entries of A are generated by a single vector, the convolution kernel h: A\_{mn} := h\_{m-n mod N}. With h = [w, x, y, z],

A = |w z y x|  
    |x w z y|  
    |y x w z|  
    |z y x w|

This means that the result of matrix multiplication v = Au has Lth column given circular convolution between h with the Lth column of u.

See http://ee.stanford.edu/~gray/toeplitz.pdf

#### Description in terms of the frequency spectrum

There is an equivalent description in terms of the [batch] spectrum H and Fourier transforms. Here we consider A.shape = [N, N] and ignore batch dimensions. Define the discrete Fourier transform (DFT) and its inverse by

DFT[ h[n] ] = H[k] := sum\_{n = 0}^{N - 1} h\_n e^{-i 2pi k n / N}  
IDFT[ H[k] ] = h[n] = N^{-1} sum\_{k = 0}^{N - 1} H\_k e^{i 2pi k n / N}

From these definitions, we see that

H[0] = sum\_{n = 0}^{N - 1} h\_n  
H[1] = "the first positive frequency"  
H[N - 1] = "the first negative frequency"

Loosely speaking, with \* element-wise multiplication, matrix multiplication is equal to the action of a Fourier multiplier: A u = IDFT[ H \* DFT[u] ]. Precisely speaking, given [N, R] matrix u, let DFT[u] be the [N, R] matrix with rth column equal to the DFT of the rth column of u. Define the IDFT similarly. Matrix multiplication may be expressed columnwise:

(A u)\_r = IDFT[ H \* (DFT[u])\_r ]

#### Operator properties deduced from the spectrum.

Letting U be the kth Euclidean basis vector, and U = IDFT[u]. The above formulas show thatA U = H\_k \* U. We conclude that the elements of H are the eigenvalues of this operator. Therefore

* This operator is positive definite if and only if Real{H} > 0.

A general property of Fourier transforms is the correspondence between Hermitian functions and real valued transforms.

Suppose H.shape = [B1,...,Bb, N]. We say that H is a Hermitian spectrum if, with % meaning modulus division,

H[..., n % N] = ComplexConjugate[ H[..., (-n) % N] ]

* This operator corresponds to a real matrix if and only if H is Hermitian.
* This operator is self-adjoint if and only if H is real.

See e.g. "Discrete-Time Signal Processing", Oppenheim and Schafer.

#### Example of a self-adjoint positive definite operator

# spectrum is real ==> operator is self-adjoint  
# spectrum is positive ==> operator is positive definite  
spectrum = [6., 4, 2]  
  
operator = LinearOperatorCirculant(spectrum)  
  
# IFFT[spectrum]  
operator.convolution\_kernel()  
==> [4 + 0j, 1 + 0.58j, 1 - 0.58j]  
  
operator.to\_dense()  
==> [[4 + 0.0j, 1 - 0.6j, 1 + 0.6j],  
     [1 + 0.6j, 4 + 0.0j, 1 - 0.6j],  
     [1 - 0.6j, 1 + 0.6j, 4 + 0.0j]]

#### Example of defining in terms of a real convolution kernel

# convolution\_kernel is real ==> spectrum is Hermitian.  
convolution\_kernel = [1., 2., 1.]]  
spectrum = tf.signal.fft(tf.cast(convolution\_kernel, tf.complex64))  
  
# spectrum is Hermitian ==> operator is real.  
# spectrum is shape [3] ==> operator is shape [3, 3]  
# We force the input/output type to be real, which allows this to operate  
# like a real matrix.  
operator = LinearOperatorCirculant(spectrum, input\_output\_dtype=tf.float32)  
  
operator.to\_dense()  
==> [[ 1, 1, 2],  
     [ 2, 1, 1],  
     [ 1, 2, 1]]

#### Example of Hermitian spectrum

# spectrum is shape [3] ==> operator is shape [3, 3]  
# spectrum is Hermitian ==> operator is real.  
spectrum = [1, 1j, -1j]  
  
operator = LinearOperatorCirculant(spectrum)  
  
operator.to\_dense()  
==> [[ 0.33 + 0j,  0.91 + 0j, -0.24 + 0j],  
     [-0.24 + 0j,  0.33 + 0j,  0.91 + 0j],  
     [ 0.91 + 0j, -0.24 + 0j,  0.33 + 0j]

#### Example of forcing real dtype when spectrum is Hermitian

# spectrum is shape [4] ==> operator is shape [4, 4]  
# spectrum is real ==> operator is self-adjoint  
# spectrum is Hermitian ==> operator is real  
# spectrum has positive real part ==> operator is positive-definite.  
spectrum = [6., 4, 2, 4]  
  
# Force the input dtype to be float32.  
# Cast the output to float32.  This is fine because the operator will be  
# real due to Hermitian spectrum.  
operator = LinearOperatorCirculant(spectrum, input\_output\_dtype=tf.float32)  
  
operator.shape  
==> [4, 4]  
  
operator.to\_dense()  
==> [[4, 1, 0, 1],  
     [1, 4, 1, 0],  
     [0, 1, 4, 1],  
     [1, 0, 1, 4]]  
  
# convolution\_kernel = tf.signal.ifft(spectrum)  
operator.convolution\_kernel()  
==> [4, 1, 0, 1]

#### Performance

Suppose operator is a LinearOperatorCirculant of shape [N, N], and x.shape = [N, R]. Then

* operator.matmul(x) is O(R\*N\*Log[N])
* operator.solve(x) is O(R\*N\*Log[N])
* operator.determinant() involves a size N reduce\_prod.

If instead operator and x have shape [B1,...,Bb, N, N] and [B1,...,Bb, N, R], every operation increases in complexity by B1\*...\*Bb.

#### Matrix property hints

This LinearOperator is initialized with boolean flags of the form is\_X, for X = non\_singular, self\_adjoint, positive\_definite, square. These have the following meaning:

* If is\_X == True, callers should expect the operator to have the property X. This is a promise that should be fulfilled, but is not a runtime assert. For example, finite floating point precision may result in these promises being violated.
* If is\_X == False, callers should expect the operator to not have X.
* If is\_X == None (the default), callers should have no expectation either way.

## \_\_init\_\_

\_\_init\_\_(  
    spectrum,  
    input\_output\_dtype=tf.dtypes.complex64,  
    is\_non\_singular=None,  
    is\_self\_adjoint=None,  
    is\_positive\_definite=None,  
    is\_square=True,  
    name='LinearOperatorCirculant'  
)

Initialize an LinearOperatorCirculant.

This LinearOperator is initialized to have shape [B1,...,Bb, N, N] by providing spectrum, a [B1,...,Bb, N] Tensor.

If input\_output\_dtype = DTYPE:

* Arguments to methods such as matmul or solve must be DTYPE.
* Values returned by all methods, such as matmul or determinant will be cast to DTYPE.

Note that if the spectrum is not Hermitian, then this operator corresponds to a complex matrix with non-zero imaginary part. In this case, setting input\_output\_dtype to a real type will forcibly cast the output to be real, resulting in incorrect results!

If on the other hand the spectrum is Hermitian, then this operator corresponds to a real-valued matrix, and setting input\_output\_dtype to a real type is fine.

#### Args:

* **spectrum**: Shape [B1,...,Bb, N] Tensor. Allowed dtypes: float16, float32, float64, complex64, complex128. Type can be different than input\_output\_dtype
* **input\_output\_dtype**: dtype for input/output.
* **is\_non\_singular**: Expect that this operator is non-singular.
* **is\_self\_adjoint**: Expect that this operator is equal to its hermitian transpose. If spectrum is real, this will always be true.
* **is\_positive\_definite**: Expect that this operator is positive definite, meaning the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive-definite. See: https://en.wikipedia.org/wiki/Positive-definite\_matrix  
  #Extension\_for\_non\_symmetric\_matrices
* **is\_square**: Expect that this operator acts like square [batch] matrices.
* **name**: A name to prepend to all ops created by this class.

## Properties

### H

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### batch\_shape

TensorShape of batch dimensions of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb]), equivalent to A.get\_shape()[:-2]

#### Returns:

TensorShape, statically determined, may be undefined.

### block\_depth

Depth of recursively defined circulant blocks defining this Operator.

With A the dense representation of this Operator,

block\_depth = 1 means A is symmetric circulant. For example,

A = |w z y x|  
    |x w z y|  
    |y x w z|  
    |z y x w|

block\_depth = 2 means A is block symmetric circulant with symemtric circulant blocks. For example, with W, X, Y, Z symmetric circulant,

A = |W Z Y X|  
    |X W Z Y|  
    |Y X W Z|  
    |Z Y X W|

block\_depth = 3 means A is block symmetric circulant with block symmetric circulant blocks.

#### Returns:

Python integer.

### block\_shape

### domain\_dimension

Dimension (in the sense of vector spaces) of the domain of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Returns:

Dimension object.

### dtype

The DType of Tensors handled by this LinearOperator.

### graph\_parents

List of graph dependencies of this LinearOperator.

### is\_non\_singular

### is\_positive\_definite

### is\_self\_adjoint

### is\_square

Return True/False depending on if this operator is square.

### name

Name prepended to all ops created by this LinearOperator.

### range\_dimension

Dimension (in the sense of vector spaces) of the range of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Returns:

Dimension object.

### shape

TensorShape of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb, M, N]), equivalent to A.get\_shape().

#### Returns:

TensorShape, statically determined, may be undefined.

### spectrum

### tensor\_rank

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

Python integer, or None if the tensor rank is undefined.

## Methods

### add\_to\_tensor

add\_to\_tensor(  
    x,  
    name='add\_to\_tensor'  
)

Add matrix represented by this operator to x. Equivalent to A + x.

#### Args:

* **x**: Tensor with same dtype and shape broadcastable to self.shape.
* **name**: A name to give this Op.

#### Returns:

A Tensor with broadcast shape and same dtype as self.

### adjoint

adjoint(name='adjoint')

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### assert\_hermitian\_spectrum

assert\_hermitian\_spectrum(name='assert\_hermitian\_spectrum')

Returns an Op that asserts this operator has Hermitian spectrum.

This operator corresponds to a real-valued matrix if and only if its spectrum is Hermitian.

#### Args:

* **name**: A name to give this Op.

#### Returns:

An Op that asserts this operator has Hermitian spectrum.

### assert\_non\_singular

assert\_non\_singular(name='assert\_non\_singular')

Returns an Op that asserts this operator is non singular.

This operator is considered non-singular if

ConditionNumber < max{100, range\_dimension, domain\_dimension} \* eps,  
eps := np.finfo(self.dtype.as\_numpy\_dtype).eps

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is singular.

### assert\_positive\_definite

assert\_positive\_definite(name='assert\_positive\_definite')

Returns an Op that asserts this operator is positive definite.

Here, positive definite means that the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive definite.

#### Args:

* **name**: A name to give this Op.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not positive definite.

### assert\_self\_adjoint

assert\_self\_adjoint(name='assert\_self\_adjoint')

Returns an Op that asserts this operator is self-adjoint.

Here we check that this operator is exactly equal to its hermitian transpose.

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not self-adjoint.

### batch\_shape\_tensor

batch\_shape\_tensor(name='batch\_shape\_tensor')

Shape of batch dimensions of this operator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb].

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### block\_shape\_tensor

block\_shape\_tensor()

Shape of the block dimensions of self.spectrum.

### cholesky

cholesky(name='cholesky')

Returns a Cholesky factor as a LinearOperator.

Given A representing this LinearOperator, if A is positive definite self-adjoint, return L, where A = L L^T, i.e. the cholesky decomposition.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the lower triangular matrix in the Cholesky decomposition.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be positive definite and self adjoint.

### convolution\_kernel

convolution\_kernel(name='convolution\_kernel')

Convolution kernel corresponding to self.spectrum.

The D dimensional DFT of this kernel is the frequency domain spectrum of this operator.

#### Args:

* **name**: A name to give this Op.

#### Returns:

Tensor with dtype self.dtype.

### determinant

determinant(name='det')

Determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### diag\_part

diag\_part(name='diag\_part')

Efficiently get the [batch] diagonal part of this operator.

If this operator has shape [B1,...,Bb, M, N], this returns a Tensor diagonal, of shape [B1,...,Bb, min(M, N)], where diagonal[b1,...,bb, i] = self.to\_dense()[b1,...,bb, i, i].

my\_operator = LinearOperatorDiag([1., 2.])  
  
# Efficiently get the diagonal  
my\_operator.diag\_part()  
==> [1., 2.]  
  
# Equivalent, but inefficient method  
tf.linalg.diag\_part(my\_operator.to\_dense())  
==> [1., 2.]

#### Args:

* **name**: A name for this Op.

#### Returns:

* **diag\_part**: A Tensor of same dtype as self.

### domain\_dimension\_tensor

domain\_dimension\_tensor(name='domain\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the domain of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### inverse

inverse(name='inverse')

Returns the Inverse of this LinearOperator.

Given A representing this LinearOperator, return a LinearOperator representing A^-1.

#### Args:

* **name**: A name scope to use for ops added by this method.

#### Returns:

LinearOperator representing inverse of this matrix.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be non\_singular.

### log\_abs\_determinant

log\_abs\_determinant(name='log\_abs\_det')

Log absolute value of determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### matmul

matmul(  
    x,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='matmul'  
)

Transform [batch] matrix x with left multiplication: x --> Ax.

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
X = ... # shape [..., N, R], batch matrix, R > 0.  
  
Y = operator.matmul(X)  
Y.shape  
==> [..., M, R]  
  
Y[..., :, r] = sum\_j A[..., :, j] X[j, r]

#### Args:

* **x**: LinearOperator or Tensor with compatible shape and same dtype as self. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **adjoint\_arg**: Python bool. If True, compute A x^H where x^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name for this Op.

#### Returns:

A LinearOperator or Tensor with shape [..., M, R] and same dtype as self.

### matvec

matvec(  
    x,  
    adjoint=False,  
    name='matvec'  
)

Transform [batch] vector x with left multiplication: x --> Ax.

# Make an operator acting like batch matric A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
  
X = ... # shape [..., N], batch vector  
  
Y = operator.matvec(X)  
Y.shape  
==> [..., M]  
  
Y[..., :] = sum\_j A[..., :, j] X[..., j]

#### Args:

* **x**: Tensor with compatible shape and same dtype as self. x is treated as a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **name**: A name for this Op.

#### Returns:

A Tensor with shape [..., M] and same dtype as self.

### range\_dimension\_tensor

range\_dimension\_tensor(name='range\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the range of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### shape\_tensor

shape\_tensor(name='shape\_tensor')

Shape of this LinearOperator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb, M, N], equivalent to tf.shape(A).

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### solve

solve(  
    rhs,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='solve'  
)

Solve (exact or approx) R (batch) systems of equations: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve R > 0 linear systems for every member of the batch.  
RHS = ... # shape [..., M, R]  
  
X = operator.solve(RHS)  
# X[..., :, r] is the solution to the r'th linear system  
# sum\_j A[..., :, j] X[..., j, r] = RHS[..., :, r]  
  
operator.matmul(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator and compatible shape. rhs is treated like a [batch] matrix meaning for every set of leading dimensions, the last two dimensions defines a matrix. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **adjoint\_arg**: Python bool. If True, solve A X = rhs^H where rhs^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N, R] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### solvevec

solvevec(  
    rhs,  
    adjoint=False,  
    name='solve'  
)

Solve single equation with best effort: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve one linear system for every member of the batch.  
RHS = ... # shape [..., M]  
  
X = operator.solvevec(RHS)  
# X is the solution to the linear system  
# sum\_j A[..., :, j] X[..., j] = RHS[..., :]  
  
operator.matvec(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator. rhs is treated like a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility regarding batch dimensions.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### tensor\_rank\_tensor

tensor\_rank\_tensor(name='tensor\_rank\_tensor')

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor, determined at runtime.

### to\_dense

to\_dense(name='to\_dense')

Return a dense (batch) matrix representing this operator.

### trace

trace(name='trace')

Trace of the linear operator, equal to sum of self.diag\_part().

If the operator is square, this is also the sum of the eigenvalues.

#### Args:

* **name**: A name for this Op.

#### Returns:

Shape [B1,...,Bb] Tensor of same dtype as self.

# tf.linalg.LinearOperatorCirculant2D

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorCirculant2D#top_of_page)
* [Class LinearOperatorCirculant2D](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorCirculant2D#class_linearoperatorcirculant2d)
  + [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorCirculant2D#aliases)
  + [Example of a self-adjoint positive definite operator](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorCirculant2D#example_of_a_self-adjoint_positive_definite_operator)
* [\_\_init\_\_](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorCirculant2D#__init__)

## Class LinearOperatorCirculant2D

LinearOperator acting like a block circulant matrix.

### Aliases:

* Class tf.compat.v1.linalg.LinearOperatorCirculant2D
* Class tf.compat.v2.linalg.LinearOperatorCirculant2D
* Class tf.linalg.LinearOperatorCirculant2D

Defined in [python/ops/linalg/linear\_operator\_circulant.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg/linear_operator_circulant.py).

This operator acts like a block circulant matrix A with shape [B1,...,Bb, N, N] for some b >= 0. The first b indices index a batch member. For every batch index (i1,...,ib), A[i1,...,ib, : :]is an N x N matrix. This matrix A is not materialized, but for purposes of broadcasting this shape will be relevant.

#### Description in terms of block circulant matrices

If A is block circulant, with block sizes N0, N1 (N0 \* N1 = N): A has a block circulant structure, composed of N0 x N0 blocks, with each block an N1 x N1 circulant matrix.

For example, with W, X, Y, Z each circulant,

A = |W Z Y X|  
    |X W Z Y|  
    |Y X W Z|  
    |Z Y X W|

Note that A itself will not in general be circulant.

#### Description in terms of the frequency spectrum

There is an equivalent description in terms of the [batch] spectrum H and Fourier transforms. Here we consider A.shape = [N, N] and ignore batch dimensions.

If H.shape = [N0, N1], (N0 \* N1 = N): Loosely speaking, matrix multiplication is equal to the action of a Fourier multiplier: A u = IDFT2[ H DFT2[u] ]. Precisely speaking, given [N, R] matrix u, let DFT2[u] be the [N0, N1, R] Tensor defined by re-shaping u to [N0, N1, R] and taking a two dimensional DFT across the first two dimensions. Let IDFT2 be the inverse of DFT2. Matrix multiplication may be expressed columnwise:

(A u)\_r = IDFT2[ H \* (DFT2[u])\_r ]

#### Operator properties deduced from the spectrum.

* This operator is positive definite if and only if Real{H} > 0.

A general property of Fourier transforms is the correspondence between Hermitian functions and real valued transforms.

Suppose H.shape = [B1,...,Bb, N0, N1], we say that H is a Hermitian spectrum if, with %indicating modulus division,

H[..., n0 % N0, n1 % N1] = ComplexConjugate[ H[..., (-n0) % N0, (-n1) % N1 ].

* This operator corresponds to a real matrix if and only if H is Hermitian.
* This operator is self-adjoint if and only if H is real.

See e.g. "Discrete-Time Signal Processing", Oppenheim and Schafer.

### Example of a self-adjoint positive definite operator

# spectrum is real ==> operator is self-adjoint  
# spectrum is positive ==> operator is positive definite  
spectrum = [[1., 2., 3.],  
            [4., 5., 6.],  
            [7., 8., 9.]]  
  
operator = LinearOperatorCirculant2D(spectrum)  
  
# IFFT[spectrum]  
operator.convolution\_kernel()  
==> [[5.0+0.0j, -0.5-.3j, -0.5+.3j],  
     [-1.5-.9j,        0,        0],  
     [-1.5+.9j,        0,        0]]  
  
operator.to\_dense()  
==> Complex self adjoint 9 x 9 matrix.

#### Example of defining in terms of a real convolution kernel,

# convolution\_kernel is real ==> spectrum is Hermitian.  
convolution\_kernel = [[1., 2., 1.], [5., -1., 1.]]  
spectrum = tf.signal.fft2d(tf.cast(convolution\_kernel, tf.complex64))  
  
# spectrum is shape [2, 3] ==> operator is shape [6, 6]  
# spectrum is Hermitian ==> operator is real.  
operator = LinearOperatorCirculant2D(spectrum, input\_output\_dtype=tf.float32)

#### Performance

Suppose operator is a LinearOperatorCirculant of shape [N, N], and x.shape = [N, R]. Then

* operator.matmul(x) is O(R\*N\*Log[N])
* operator.solve(x) is O(R\*N\*Log[N])
* operator.determinant() involves a size N reduce\_prod.

If instead operator and x have shape [B1,...,Bb, N, N] and [B1,...,Bb, N, R], every operation increases in complexity by B1\*...\*Bb.

#### Matrix property hints

This LinearOperator is initialized with boolean flags of the form is\_X, for X = non\_singular, self\_adjoint, positive\_definite, square. These have the following meaning \* If is\_X == True, callers should expect the operator to have the property X. This is a promise that should be fulfilled, but is not a runtime assert. For example, finite floating point precision may result in these promises being violated. \* If is\_X == False, callers should expect the operator to not have X. \* If is\_X == None (the default), callers should have no expectation either way.

## \_\_init\_\_

\_\_init\_\_(  
    spectrum,  
    input\_output\_dtype=tf.dtypes.complex64,  
    is\_non\_singular=None,  
    is\_self\_adjoint=None,  
    is\_positive\_definite=None,  
    is\_square=True,  
    name='LinearOperatorCirculant2D'  
)

Initialize an LinearOperatorCirculant2D.

This LinearOperator is initialized to have shape [B1,...,Bb, N, N] by providing spectrum, a [B1,...,Bb, N0, N1] Tensor with N0\*N1 = N.

If input\_output\_dtype = DTYPE:

* Arguments to methods such as matmul or solve must be DTYPE.
* Values returned by all methods, such as matmul or determinant will be cast to DTYPE.

Note that if the spectrum is not Hermitian, then this operator corresponds to a complex matrix with non-zero imaginary part. In this case, setting input\_output\_dtype to a real type will forcibly cast the output to be real, resulting in incorrect results!

If on the other hand the spectrum is Hermitian, then this operator corresponds to a real-valued matrix, and setting input\_output\_dtype to a real type is fine.

#### Args:

* **spectrum**: Shape [B1,...,Bb, N] Tensor. Allowed dtypes: float16, float32, float64, complex64, complex128. Type can be different than input\_output\_dtype
* **input\_output\_dtype**: dtype for input/output.
* **is\_non\_singular**: Expect that this operator is non-singular.
* **is\_self\_adjoint**: Expect that this operator is equal to its hermitian transpose. If spectrum is real, this will always be true.
* **is\_positive\_definite**: Expect that this operator is positive definite, meaning the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive-definite. See: https://en.wikipedia.org/wiki/Positive-definite\_matrix  
  #Extension\_for\_non\_symmetric\_matrices
* **is\_square**: Expect that this operator acts like square [batch] matrices.
* **name**: A name to prepend to all ops created by this class.

## Properties

### H

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### batch\_shape

TensorShape of batch dimensions of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb]), equivalent to A.get\_shape()[:-2]

#### Returns:

TensorShape, statically determined, may be undefined.

### block\_depth

Depth of recursively defined circulant blocks defining this Operator.

With A the dense representation of this Operator,

block\_depth = 1 means A is symmetric circulant. For example,

A = |w z y x|  
    |x w z y|  
    |y x w z|  
    |z y x w|

block\_depth = 2 means A is block symmetric circulant with symemtric circulant blocks. For example, with W, X, Y, Z symmetric circulant,

A = |W Z Y X|  
    |X W Z Y|  
    |Y X W Z|  
    |Z Y X W|

block\_depth = 3 means A is block symmetric circulant with block symmetric circulant blocks.

#### Returns:

Python integer.

### block\_shape

### domain\_dimension

Dimension (in the sense of vector spaces) of the domain of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Returns:

Dimension object.

### dtype

The DType of Tensors handled by this LinearOperator.

### graph\_parents

List of graph dependencies of this LinearOperator.

### is\_non\_singular

### is\_positive\_definite

### is\_self\_adjoint

### is\_square

Return True/False depending on if this operator is square.

### name

Name prepended to all ops created by this LinearOperator.

### range\_dimension

Dimension (in the sense of vector spaces) of the range of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Returns:

Dimension object.

### shape

TensorShape of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb, M, N]), equivalent to A.get\_shape().

#### Returns:

TensorShape, statically determined, may be undefined.

### spectrum

### tensor\_rank

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

Python integer, or None if the tensor rank is undefined.

## Methods

### add\_to\_tensor

add\_to\_tensor(  
    x,  
    name='add\_to\_tensor'  
)

Add matrix represented by this operator to x. Equivalent to A + x.

#### Args:

* **x**: Tensor with same dtype and shape broadcastable to self.shape.
* **name**: A name to give this Op.

#### Returns:

A Tensor with broadcast shape and same dtype as self.

### adjoint

adjoint(name='adjoint')

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### assert\_hermitian\_spectrum

assert\_hermitian\_spectrum(name='assert\_hermitian\_spectrum')

Returns an Op that asserts this operator has Hermitian spectrum.

This operator corresponds to a real-valued matrix if and only if its spectrum is Hermitian.

#### Args:

* **name**: A name to give this Op.

#### Returns:

An Op that asserts this operator has Hermitian spectrum.

### assert\_non\_singular

assert\_non\_singular(name='assert\_non\_singular')

Returns an Op that asserts this operator is non singular.

This operator is considered non-singular if

ConditionNumber < max{100, range\_dimension, domain\_dimension} \* eps,  
eps := np.finfo(self.dtype.as\_numpy\_dtype).eps

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is singular.

### assert\_positive\_definite

assert\_positive\_definite(name='assert\_positive\_definite')

Returns an Op that asserts this operator is positive definite.

Here, positive definite means that the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive definite.

#### Args:

* **name**: A name to give this Op.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not positive definite.

### assert\_self\_adjoint

assert\_self\_adjoint(name='assert\_self\_adjoint')

Returns an Op that asserts this operator is self-adjoint.

Here we check that this operator is exactly equal to its hermitian transpose.

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not self-adjoint.

### batch\_shape\_tensor

batch\_shape\_tensor(name='batch\_shape\_tensor')

Shape of batch dimensions of this operator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb].

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### block\_shape\_tensor

block\_shape\_tensor()

Shape of the block dimensions of self.spectrum.

### cholesky

cholesky(name='cholesky')

Returns a Cholesky factor as a LinearOperator.

Given A representing this LinearOperator, if A is positive definite self-adjoint, return L, where A = L L^T, i.e. the cholesky decomposition.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the lower triangular matrix in the Cholesky decomposition.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be positive definite and self adjoint.

### convolution\_kernel

convolution\_kernel(name='convolution\_kernel')

Convolution kernel corresponding to self.spectrum.

The D dimensional DFT of this kernel is the frequency domain spectrum of this operator.

#### Args:

* **name**: A name to give this Op.

#### Returns:

Tensor with dtype self.dtype.

### determinant

determinant(name='det')

Determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### diag\_part

diag\_part(name='diag\_part')

Efficiently get the [batch] diagonal part of this operator.

If this operator has shape [B1,...,Bb, M, N], this returns a Tensor diagonal, of shape [B1,...,Bb, min(M, N)], where diagonal[b1,...,bb, i] = self.to\_dense()[b1,...,bb, i, i].

my\_operator = LinearOperatorDiag([1., 2.])  
  
# Efficiently get the diagonal  
my\_operator.diag\_part()  
==> [1., 2.]  
  
# Equivalent, but inefficient method  
tf.linalg.diag\_part(my\_operator.to\_dense())  
==> [1., 2.]

#### Args:

* **name**: A name for this Op.

#### Returns:

* **diag\_part**: A Tensor of same dtype as self.

### domain\_dimension\_tensor

domain\_dimension\_tensor(name='domain\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the domain of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### inverse

inverse(name='inverse')

Returns the Inverse of this LinearOperator.

Given A representing this LinearOperator, return a LinearOperator representing A^-1.

#### Args:

* **name**: A name scope to use for ops added by this method.

#### Returns:

LinearOperator representing inverse of this matrix.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be non\_singular.

### log\_abs\_determinant

log\_abs\_determinant(name='log\_abs\_det')

Log absolute value of determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### matmul

matmul(  
    x,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='matmul'  
)

Transform [batch] matrix x with left multiplication: x --> Ax.

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
X = ... # shape [..., N, R], batch matrix, R > 0.  
  
Y = operator.matmul(X)  
Y.shape  
==> [..., M, R]  
  
Y[..., :, r] = sum\_j A[..., :, j] X[j, r]

#### Args:

* **x**: LinearOperator or Tensor with compatible shape and same dtype as self. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **adjoint\_arg**: Python bool. If True, compute A x^H where x^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name for this Op.

#### Returns:

A LinearOperator or Tensor with shape [..., M, R] and same dtype as self.

### matvec

matvec(  
    x,  
    adjoint=False,  
    name='matvec'  
)

Transform [batch] vector x with left multiplication: x --> Ax.

# Make an operator acting like batch matric A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
  
X = ... # shape [..., N], batch vector  
  
Y = operator.matvec(X)  
Y.shape  
==> [..., M]  
  
Y[..., :] = sum\_j A[..., :, j] X[..., j]

#### Args:

* **x**: Tensor with compatible shape and same dtype as self. x is treated as a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **name**: A name for this Op.

#### Returns:

A Tensor with shape [..., M] and same dtype as self.

### range\_dimension\_tensor

range\_dimension\_tensor(name='range\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the range of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### shape\_tensor

shape\_tensor(name='shape\_tensor')

Shape of this LinearOperator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb, M, N], equivalent to tf.shape(A).

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### solve

solve(  
    rhs,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='solve'  
)

Solve (exact or approx) R (batch) systems of equations: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve R > 0 linear systems for every member of the batch.  
RHS = ... # shape [..., M, R]  
  
X = operator.solve(RHS)  
# X[..., :, r] is the solution to the r'th linear system  
# sum\_j A[..., :, j] X[..., j, r] = RHS[..., :, r]  
  
operator.matmul(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator and compatible shape. rhs is treated like a [batch] matrix meaning for every set of leading dimensions, the last two dimensions defines a matrix. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **adjoint\_arg**: Python bool. If True, solve A X = rhs^H where rhs^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N, R] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### solvevec

solvevec(  
    rhs,  
    adjoint=False,  
    name='solve'  
)

Solve single equation with best effort: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve one linear system for every member of the batch.  
RHS = ... # shape [..., M]  
  
X = operator.solvevec(RHS)  
# X is the solution to the linear system  
# sum\_j A[..., :, j] X[..., j] = RHS[..., :]  
  
operator.matvec(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator. rhs is treated like a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility regarding batch dimensions.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### tensor\_rank\_tensor

tensor\_rank\_tensor(name='tensor\_rank\_tensor')

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor, determined at runtime.

### to\_dense

to\_dense(name='to\_dense')

Return a dense (batch) matrix representing this operator.

### trace

trace(name='trace')

Trace of the linear operator, equal to sum of self.diag\_part().

If the operator is square, this is also the sum of the eigenvalues.

#### Args:

* **name**: A name for this Op.

#### Returns:

Shape [B1,...,Bb] Tensor of same dtype as self.

# tf.linalg.LinearOperatorCirculant3D

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorCirculant3D#top_of_page)
* [Class LinearOperatorCirculant3D](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorCirculant3D#class_linearoperatorcirculant3d)
  + [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorCirculant3D#aliases)
  + [Examples](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorCirculant3D#examples)
* [\_\_init\_\_](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorCirculant3D#__init__)

## Class LinearOperatorCirculant3D

LinearOperator acting like a nested block circulant matrix.

### Aliases:

* Class tf.compat.v1.linalg.LinearOperatorCirculant3D
* Class tf.compat.v2.linalg.LinearOperatorCirculant3D
* Class tf.linalg.LinearOperatorCirculant3D

Defined in [python/ops/linalg/linear\_operator\_circulant.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg/linear_operator_circulant.py).

This operator acts like a block circulant matrix A with shape [B1,...,Bb, N, N] for some b >= 0. The first b indices index a batch member. For every batch index (i1,...,ib), A[i1,...,ib, : :]is an N x N matrix. This matrix A is not materialized, but for purposes of broadcasting this shape will be relevant.

#### Description in terms of block circulant matrices

If A is nested block circulant, with block sizes N0, N1, N2 (N0 \* N1 \* N2 = N): A has a block structure, composed of N0 x N0 blocks, with each block an N1 x N1 block circulant matrix.

For example, with W, X, Y, Z each block circulant,

A = |W Z Y X|  
    |X W Z Y|  
    |Y X W Z|  
    |Z Y X W|

Note that A itself will not in general be circulant.

#### Description in terms of the frequency spectrum

There is an equivalent description in terms of the [batch] spectrum H and Fourier transforms. Here we consider A.shape = [N, N] and ignore batch dimensions.

If H.shape = [N0, N1, N2], (N0 \* N1 \* N2 = N): Loosely speaking, matrix multiplication is equal to the action of a Fourier multiplier: A u = IDFT3[ H DFT3[u] ]. Precisely speaking, given [N, R]matrix u, let DFT3[u] be the [N0, N1, N2, R] Tensor defined by re-shaping u to [N0, N1, N2, R] and taking a three dimensional DFT across the first three dimensions. Let IDFT3 be the inverse of DFT3. Matrix multiplication may be expressed columnwise:

(A u)\_r = IDFT3[ H \* (DFT3[u])\_r ]

#### Operator properties deduced from the spectrum.

* This operator is positive definite if and only if Real{H} > 0.

A general property of Fourier transforms is the correspondence between Hermitian functions and real valued transforms.

Suppose H.shape = [B1,...,Bb, N0, N1, N2], we say that H is a Hermitian spectrum if, with %meaning modulus division,

H[..., n0 % N0, n1 % N1, n2 % N2]  
  = ComplexConjugate[ H[..., (-n0) % N0, (-n1) % N1, (-n2) % N2] ].

* This operator corresponds to a real matrix if and only if H is Hermitian.
* This operator is self-adjoint if and only if H is real.

See e.g. "Discrete-Time Signal Processing", Oppenheim and Schafer.

### Examples

See LinearOperatorCirculant and LinearOperatorCirculant2D for examples.

#### Performance

Suppose operator is a LinearOperatorCirculant of shape [N, N], and x.shape = [N, R]. Then

* operator.matmul(x) is O(R\*N\*Log[N])
* operator.solve(x) is O(R\*N\*Log[N])
* operator.determinant() involves a size N reduce\_prod.

If instead operator and x have shape [B1,...,Bb, N, N] and [B1,...,Bb, N, R], every operation increases in complexity by B1\*...\*Bb.

#### Matrix property hints

This LinearOperator is initialized with boolean flags of the form is\_X, for X = non\_singular, self\_adjoint, positive\_definite, square. These have the following meaning \* If is\_X == True, callers should expect the operator to have the property X. This is a promise that should be fulfilled, but is not a runtime assert. For example, finite floating point precision may result in these promises being violated. \* If is\_X == False, callers should expect the operator to not have X. \* If is\_X == None (the default), callers should have no expectation either way.

## \_\_init\_\_

\_\_init\_\_(  
    spectrum,  
    input\_output\_dtype=tf.dtypes.complex64,  
    is\_non\_singular=None,  
    is\_self\_adjoint=None,  
    is\_positive\_definite=None,  
    is\_square=True,  
    name='LinearOperatorCirculant3D'  
)

Initialize an LinearOperatorCirculant.

This LinearOperator is initialized to have shape [B1,...,Bb, N, N] by providing spectrum, a [B1,...,Bb, N0, N1, N2] Tensor with N0\*N1\*N2 = N.

If input\_output\_dtype = DTYPE:

* Arguments to methods such as matmul or solve must be DTYPE.
* Values returned by all methods, such as matmul or determinant will be cast to DTYPE.

Note that if the spectrum is not Hermitian, then this operator corresponds to a complex matrix with non-zero imaginary part. In this case, setting input\_output\_dtype to a real type will forcibly cast the output to be real, resulting in incorrect results!

If on the other hand the spectrum is Hermitian, then this operator corresponds to a real-valued matrix, and setting input\_output\_dtype to a real type is fine.

#### Args:

* **spectrum**: Shape [B1,...,Bb, N] Tensor. Allowed dtypes: float16, float32, float64, complex64, complex128. Type can be different than input\_output\_dtype
* **input\_output\_dtype**: dtype for input/output.
* **is\_non\_singular**: Expect that this operator is non-singular.
* **is\_self\_adjoint**: Expect that this operator is equal to its hermitian transpose. If spectrum is real, this will always be true.
* **is\_positive\_definite**: Expect that this operator is positive definite, meaning the real part of all eigenvalues is positive. We do not require the operator to be self-adjoint to be positive-definite. See: https://en.wikipedia.org/wiki/Positive-definite\_matrix #Extension\_for\_non\_symmetric\_matrices
* **is\_square**: Expect that this operator acts like square [batch] matrices.
* **name**: A name to prepend to all ops created by this class.

## Properties

### H

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### batch\_shape

TensorShape of batch dimensions of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb]), equivalent to A.get\_shape()[:-2]

#### Returns:

TensorShape, statically determined, may be undefined.

### block\_depth

Depth of recursively defined circulant blocks defining this Operator.

With A the dense representation of this Operator,

block\_depth = 1 means A is symmetric circulant. For example,

A = |w z y x|  
    |x w z y|  
    |y x w z|  
    |z y x w|

block\_depth = 2 means A is block symmetric circulant with symemtric circulant blocks. For example, with W, X, Y, Z symmetric circulant,

A = |W Z Y X|  
    |X W Z Y|  
    |Y X W Z|  
    |Z Y X W|

block\_depth = 3 means A is block symmetric circulant with block symmetric circulant blocks.

#### Returns:

Python integer.

### block\_shape

### domain\_dimension

Dimension (in the sense of vector spaces) of the domain of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Returns:

Dimension object.

### dtype

The DType of Tensors handled by this LinearOperator.

### graph\_parents

List of graph dependencies of this LinearOperator.

### is\_non\_singular

### is\_positive\_definite

### is\_self\_adjoint

### is\_square

Return True/False depending on if this operator is square.

### name

Name prepended to all ops created by this LinearOperator.

### range\_dimension

Dimension (in the sense of vector spaces) of the range of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Returns:

Dimension object.

### shape

TensorShape of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb, M, N]), equivalent to A.get\_shape().

#### Returns:

TensorShape, statically determined, may be undefined.

### spectrum

### tensor\_rank

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

Python integer, or None if the tensor rank is undefined.

## Methods

### add\_to\_tensor

add\_to\_tensor(  
    x,  
    name='add\_to\_tensor'  
)

Add matrix represented by this operator to x. Equivalent to A + x.

#### Args:

* **x**: Tensor with same dtype and shape broadcastable to self.shape.
* **name**: A name to give this Op.

#### Returns:

A Tensor with broadcast shape and same dtype as self.

### adjoint

adjoint(name='adjoint')

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### assert\_hermitian\_spectrum

assert\_hermitian\_spectrum(name='assert\_hermitian\_spectrum')

Returns an Op that asserts this operator has Hermitian spectrum.

This operator corresponds to a real-valued matrix if and only if its spectrum is Hermitian.

#### Args:

* **name**: A name to give this Op.

#### Returns:

An Op that asserts this operator has Hermitian spectrum.

### assert\_non\_singular

assert\_non\_singular(name='assert\_non\_singular')

Returns an Op that asserts this operator is non singular.

This operator is considered non-singular if

ConditionNumber < max{100, range\_dimension, domain\_dimension} \* eps,  
eps := np.finfo(self.dtype.as\_numpy\_dtype).eps

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is singular.

### assert\_positive\_definite

assert\_positive\_definite(name='assert\_positive\_definite')

Returns an Op that asserts this operator is positive definite.

Here, positive definite means that the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive definite.

#### Args:

* **name**: A name to give this Op.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not positive definite.

### assert\_self\_adjoint

assert\_self\_adjoint(name='assert\_self\_adjoint')

Returns an Op that asserts this operator is self-adjoint.

Here we check that this operator is exactly equal to its hermitian transpose.

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not self-adjoint.

### batch\_shape\_tensor

batch\_shape\_tensor(name='batch\_shape\_tensor')

Shape of batch dimensions of this operator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb].

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### block\_shape\_tensor

block\_shape\_tensor()

Shape of the block dimensions of self.spectrum.

### cholesky

cholesky(name='cholesky')

Returns a Cholesky factor as a LinearOperator.

Given A representing this LinearOperator, if A is positive definite self-adjoint, return L, where A = L L^T, i.e. the cholesky decomposition.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the lower triangular matrix in the Cholesky decomposition.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be positive definite and self adjoint.

### convolution\_kernel

convolution\_kernel(name='convolution\_kernel')

Convolution kernel corresponding to self.spectrum.

The D dimensional DFT of this kernel is the frequency domain spectrum of this operator.

#### Args:

* **name**: A name to give this Op.

#### Returns:

Tensor with dtype self.dtype.

### determinant

determinant(name='det')

Determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### diag\_part

diag\_part(name='diag\_part')

Efficiently get the [batch] diagonal part of this operator.

If this operator has shape [B1,...,Bb, M, N], this returns a Tensor diagonal, of shape [B1,...,Bb, min(M, N)], where diagonal[b1,...,bb, i] = self.to\_dense()[b1,...,bb, i, i].

my\_operator = LinearOperatorDiag([1., 2.])  
  
# Efficiently get the diagonal  
my\_operator.diag\_part()  
==> [1., 2.]  
  
# Equivalent, but inefficient method  
tf.linalg.diag\_part(my\_operator.to\_dense())  
==> [1., 2.]

#### Args:

* **name**: A name for this Op.

#### Returns:

* **diag\_part**: A Tensor of same dtype as self.

### domain\_dimension\_tensor

domain\_dimension\_tensor(name='domain\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the domain of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### inverse

inverse(name='inverse')

Returns the Inverse of this LinearOperator.

Given A representing this LinearOperator, return a LinearOperator representing A^-1.

#### Args:

* **name**: A name scope to use for ops added by this method.

#### Returns:

LinearOperator representing inverse of this matrix.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be non\_singular.

### log\_abs\_determinant

log\_abs\_determinant(name='log\_abs\_det')

Log absolute value of determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### matmul

matmul(  
    x,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='matmul'  
)

Transform [batch] matrix x with left multiplication: x --> Ax.

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
X = ... # shape [..., N, R], batch matrix, R > 0.  
  
Y = operator.matmul(X)  
Y.shape  
==> [..., M, R]  
  
Y[..., :, r] = sum\_j A[..., :, j] X[j, r]

#### Args:

* **x**: LinearOperator or Tensor with compatible shape and same dtype as self. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **adjoint\_arg**: Python bool. If True, compute A x^H where x^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name for this Op.

#### Returns:

A LinearOperator or Tensor with shape [..., M, R] and same dtype as self.

### matvec

matvec(  
    x,  
    adjoint=False,  
    name='matvec'  
)

Transform [batch] vector x with left multiplication: x --> Ax.

# Make an operator acting like batch matric A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
  
X = ... # shape [..., N], batch vector  
  
Y = operator.matvec(X)  
Y.shape  
==> [..., M]  
  
Y[..., :] = sum\_j A[..., :, j] X[..., j]

#### Args:

* **x**: Tensor with compatible shape and same dtype as self. x is treated as a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **name**: A name for this Op.

#### Returns:

A Tensor with shape [..., M] and same dtype as self.

### range\_dimension\_tensor

range\_dimension\_tensor(name='range\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the range of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### shape\_tensor

shape\_tensor(name='shape\_tensor')

Shape of this LinearOperator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb, M, N], equivalent to tf.shape(A).

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### solve

solve(  
    rhs,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='solve'  
)

Solve (exact or approx) R (batch) systems of equations: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve R > 0 linear systems for every member of the batch.  
RHS = ... # shape [..., M, R]  
  
X = operator.solve(RHS)  
# X[..., :, r] is the solution to the r'th linear system  
# sum\_j A[..., :, j] X[..., j, r] = RHS[..., :, r]  
  
operator.matmul(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator and compatible shape. rhs is treated like a [batch] matrix meaning for every set of leading dimensions, the last two dimensions defines a matrix. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **adjoint\_arg**: Python bool. If True, solve A X = rhs^H where rhs^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N, R] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### solvevec

solvevec(  
    rhs,  
    adjoint=False,  
    name='solve'  
)

Solve single equation with best effort: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve one linear system for every member of the batch.  
RHS = ... # shape [..., M]  
  
X = operator.solvevec(RHS)  
# X is the solution to the linear system  
# sum\_j A[..., :, j] X[..., j] = RHS[..., :]  
  
operator.matvec(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator. rhs is treated like a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility regarding batch dimensions.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### tensor\_rank\_tensor

tensor\_rank\_tensor(name='tensor\_rank\_tensor')

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor, determined at runtime.

### to\_dense

to\_dense(name='to\_dense')

Return a dense (batch) matrix representing this operator.

### trace

trace(name='trace')

Trace of the linear operator, equal to sum of self.diag\_part().

If the operator is square, this is also the sum of the eigenvalues.

#### Args:

* **name**: A name for this Op.

#### Returns:

Shape [B1,...,Bb] Tensor of same dtype as self.

# tf.linalg.LinearOperatorComposition

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorComposition#top_of_page)
* [Class LinearOperatorComposition](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorComposition#class_linearoperatorcomposition)
  + [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorComposition#aliases)
* [\_\_init\_\_](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorComposition#__init__)
* [Properties](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorComposition#properties)

## Class LinearOperatorComposition

Composes one or more LinearOperators.

Inherits From: [LinearOperator](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperator)

### Aliases:

* Class tf.compat.v1.linalg.LinearOperatorComposition
* Class tf.compat.v2.linalg.LinearOperatorComposition
* Class tf.linalg.LinearOperatorComposition

Defined in [python/ops/linalg/linear\_operator\_composition.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg/linear_operator_composition.py).

This operator composes one or more linear operators [op1,...,opJ], building a new LinearOperator with action defined by:

op\_composed(x) := op1(op2(...(opJ(x)...))

If opj acts like [batch] matrix Aj, then op\_composed acts like the [batch] matrix formed with the multiplication A1 A2...AJ.

If opj has shape batch\_shape\_j + [M\_j, N\_j], then we must have N\_j = M\_{j+1}, in which case the composed operator has shape equal to broadcast\_batch\_shape + [M\_1, N\_J], where broadcast\_batch\_shape is the mutual broadcast of batch\_shape\_j, j = 1,...,J, assuming the intermediate batch shapes broadcast. Even if the composed shape is well defined, the composed operator's methods may fail due to lack of broadcasting ability in the defining operators' methods.

# Create a 2 x 2 linear operator composed of two 2 x 2 operators.  
operator\_1 = LinearOperatorFullMatrix([[1., 2.], [3., 4.]])  
operator\_2 = LinearOperatorFullMatrix([[1., 0.], [0., 1.]])  
operator = LinearOperatorComposition([operator\_1, operator\_2])  
  
operator.to\_dense()  
==> [[1., 2.]  
     [3., 4.]]  
  
operator.shape  
==> [2, 2]  
  
operator.log\_abs\_determinant()  
==> scalar Tensor  
  
x = ... Shape [2, 4] Tensor  
operator.matmul(x)  
==> Shape [2, 4] Tensor  
  
# Create a [2, 3] batch of 4 x 5 linear operators.  
matrix\_45 = tf.random.normal(shape=[2, 3, 4, 5])  
operator\_45 = LinearOperatorFullMatrix(matrix)  
  
# Create a [2, 3] batch of 5 x 6 linear operators.  
matrix\_56 = tf.random.normal(shape=[2, 3, 5, 6])  
operator\_56 = LinearOperatorFullMatrix(matrix\_56)  
  
# Compose to create a [2, 3] batch of 4 x 6 operators.  
operator\_46 = LinearOperatorComposition([operator\_45, operator\_56])  
  
# Create a shape [2, 3, 6, 2] vector.  
x = tf.random.normal(shape=[2, 3, 6, 2])  
operator.matmul(x)  
==> Shape [2, 3, 4, 2] Tensor

#### Performance

The performance of LinearOperatorComposition on any operation is equal to the sum of the individual operators' operations.

#### Matrix property hints

This LinearOperator is initialized with boolean flags of the form is\_X, for X = non\_singular, self\_adjoint, positive\_definite, square. These have the following meaning:

* If is\_X == True, callers should expect the operator to have the property X. This is a promise that should be fulfilled, but is not a runtime assert. For example, finite floating point precision may result in these promises being violated.
* If is\_X == False, callers should expect the operator to not have X.
* If is\_X == None (the default), callers should have no expectation either way.

## \_\_init\_\_

\_\_init\_\_(  
    operators,  
    is\_non\_singular=None,  
    is\_self\_adjoint=None,  
    is\_positive\_definite=None,  
    is\_square=None,  
    name=None  
)

Initialize a LinearOperatorComposition.

LinearOperatorComposition is initialized with a list of operators [op\_1,...,op\_J]. For the matmul method to be well defined, the composition op\_i.matmul(op\_{i+1}(x)) must be defined. Other methods have similar constraints.

#### Args:

* **operators**: Iterable of LinearOperator objects, each with the same dtype and composable shape.
* **is\_non\_singular**: Expect that this operator is non-singular.
* **is\_self\_adjoint**: Expect that this operator is equal to its hermitian transpose.
* **is\_positive\_definite**: Expect that this operator is positive definite, meaning the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive-definite. See: https://en.wikipedia.org/wiki/Positive-definite\_matrix#Extension\_for\_non-symmetric\_matrices
* **is\_square**: Expect that this operator acts like square [batch] matrices.
* **name**: A name for this LinearOperator. Default is the individual operators names joined with \_o\_.

#### Raises:

* **TypeError**: If all operators do not have the same dtype.
* **ValueError**: If operators is empty.

## Properties

### H

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### batch\_shape

TensorShape of batch dimensions of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb]), equivalent to A.get\_shape()[:-2]

#### Returns:

TensorShape, statically determined, may be undefined.

### domain\_dimension

Dimension (in the sense of vector spaces) of the domain of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Returns:

Dimension object.

### dtype

The DType of Tensors handled by this LinearOperator.

### graph\_parents

List of graph dependencies of this LinearOperator.

### is\_non\_singular

### is\_positive\_definite

### is\_self\_adjoint

### is\_square

Return True/False depending on if this operator is square.

### name

Name prepended to all ops created by this LinearOperator.

### operators

### range\_dimension

Dimension (in the sense of vector spaces) of the range of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Returns:

Dimension object.

### shape

TensorShape of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb, M, N]), equivalent to A.get\_shape().

#### Returns:

TensorShape, statically determined, may be undefined.

### tensor\_rank

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

Python integer, or None if the tensor rank is undefined.

## Methods

### add\_to\_tensor

add\_to\_tensor(  
    x,  
    name='add\_to\_tensor'  
)

Add matrix represented by this operator to x. Equivalent to A + x.

#### Args:

* **x**: Tensor with same dtype and shape broadcastable to self.shape.
* **name**: A name to give this Op.

#### Returns:

A Tensor with broadcast shape and same dtype as self.

### adjoint

adjoint(name='adjoint')

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### assert\_non\_singular

assert\_non\_singular(name='assert\_non\_singular')

Returns an Op that asserts this operator is non singular.

This operator is considered non-singular if

ConditionNumber < max{100, range\_dimension, domain\_dimension} \* eps,  
eps := np.finfo(self.dtype.as\_numpy\_dtype).eps

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is singular.

### assert\_positive\_definite

assert\_positive\_definite(name='assert\_positive\_definite')

Returns an Op that asserts this operator is positive definite.

Here, positive definite means that the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive definite.

#### Args:

* **name**: A name to give this Op.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not positive definite.

### assert\_self\_adjoint

assert\_self\_adjoint(name='assert\_self\_adjoint')

Returns an Op that asserts this operator is self-adjoint.

Here we check that this operator is exactly equal to its hermitian transpose.

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not self-adjoint.

### batch\_shape\_tensor

batch\_shape\_tensor(name='batch\_shape\_tensor')

Shape of batch dimensions of this operator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb].

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### cholesky

cholesky(name='cholesky')

Returns a Cholesky factor as a LinearOperator.

Given A representing this LinearOperator, if A is positive definite self-adjoint, return L, where A = L L^T, i.e. the cholesky decomposition.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the lower triangular matrix in the Cholesky decomposition.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be positive definite and self adjoint.

### determinant

determinant(name='det')

Determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### diag\_part

diag\_part(name='diag\_part')

Efficiently get the [batch] diagonal part of this operator.

If this operator has shape [B1,...,Bb, M, N], this returns a Tensor diagonal, of shape [B1,...,Bb, min(M, N)], where diagonal[b1,...,bb, i] = self.to\_dense()[b1,...,bb, i, i].

my\_operator = LinearOperatorDiag([1., 2.])  
  
# Efficiently get the diagonal  
my\_operator.diag\_part()  
==> [1., 2.]  
  
# Equivalent, but inefficient method  
tf.linalg.diag\_part(my\_operator.to\_dense())  
==> [1., 2.]

#### Args:

* **name**: A name for this Op.

#### Returns:

* **diag\_part**: A Tensor of same dtype as self.

### domain\_dimension\_tensor

domain\_dimension\_tensor(name='domain\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the domain of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### inverse

inverse(name='inverse')

Returns the Inverse of this LinearOperator.

Given A representing this LinearOperator, return a LinearOperator representing A^-1.

#### Args:

* **name**: A name scope to use for ops added by this method.

#### Returns:

LinearOperator representing inverse of this matrix.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be non\_singular.

### log\_abs\_determinant

log\_abs\_determinant(name='log\_abs\_det')

Log absolute value of determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### matmul

matmul(  
    x,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='matmul'  
)

Transform [batch] matrix x with left multiplication: x --> Ax.

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
X = ... # shape [..., N, R], batch matrix, R > 0.  
  
Y = operator.matmul(X)  
Y.shape  
==> [..., M, R]  
  
Y[..., :, r] = sum\_j A[..., :, j] X[j, r]

#### Args:

* **x**: LinearOperator or Tensor with compatible shape and same dtype as self. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **adjoint\_arg**: Python bool. If True, compute A x^H where x^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name for this Op.

#### Returns:

A LinearOperator or Tensor with shape [..., M, R] and same dtype as self.

### matvec

matvec(  
    x,  
    adjoint=False,  
    name='matvec'  
)

Transform [batch] vector x with left multiplication: x --> Ax.

# Make an operator acting like batch matric A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
  
X = ... # shape [..., N], batch vector  
  
Y = operator.matvec(X)  
Y.shape  
==> [..., M]  
  
Y[..., :] = sum\_j A[..., :, j] X[..., j]

#### Args:

* **x**: Tensor with compatible shape and same dtype as self. x is treated as a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **name**: A name for this Op.

#### Returns:

A Tensor with shape [..., M] and same dtype as self.

### range\_dimension\_tensor

range\_dimension\_tensor(name='range\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the range of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### shape\_tensor

shape\_tensor(name='shape\_tensor')

Shape of this LinearOperator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb, M, N], equivalent to tf.shape(A).

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### solve

solve(  
    rhs,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='solve'  
)

Solve (exact or approx) R (batch) systems of equations: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve R > 0 linear systems for every member of the batch.  
RHS = ... # shape [..., M, R]  
  
X = operator.solve(RHS)  
# X[..., :, r] is the solution to the r'th linear system  
# sum\_j A[..., :, j] X[..., j, r] = RHS[..., :, r]  
  
operator.matmul(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator and compatible shape. rhs is treated like a [batch] matrix meaning for every set of leading dimensions, the last two dimensions defines a matrix. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **adjoint\_arg**: Python bool. If True, solve A X = rhs^H where rhs^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N, R] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### solvevec

solvevec(  
    rhs,  
    adjoint=False,  
    name='solve'  
)

Solve single equation with best effort: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve one linear system for every member of the batch.  
RHS = ... # shape [..., M]  
  
X = operator.solvevec(RHS)  
# X is the solution to the linear system  
# sum\_j A[..., :, j] X[..., j] = RHS[..., :]  
  
operator.matvec(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator. rhs is treated like a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility regarding batch dimensions.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### tensor\_rank\_tensor

tensor\_rank\_tensor(name='tensor\_rank\_tensor')

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor, determined at runtime.

### to\_dense

to\_dense(name='to\_dense')

Return a dense (batch) matrix representing this operator.

### trace

trace(name='trace')

Trace of the linear operator, equal to sum of self.diag\_part().

If the operator is square, this is also the sum of the eigenvalues.

#### Args:

* **name**: A name for this Op.

#### Returns:

Shape [B1,...,Bb] Tensor of same dtype as self.

# tf.linalg.LinearOperatorDiag

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorDiag#top_of_page)
* [Class LinearOperatorDiag](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorDiag#class_linearoperatordiag)
  + [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorDiag#aliases)
* [\_\_init\_\_](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorDiag#__init__)
* [Properties](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorDiag#properties)

## Class LinearOperatorDiag

LinearOperator acting like a [batch] square diagonal matrix.

Inherits From: [LinearOperator](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperator)

### Aliases:

* Class tf.compat.v1.linalg.LinearOperatorDiag
* Class tf.compat.v2.linalg.LinearOperatorDiag
* Class tf.linalg.LinearOperatorDiag

Defined in [python/ops/linalg/linear\_operator\_diag.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg/linear_operator_diag.py).

This operator acts like a [batch] diagonal matrix A with shape [B1,...,Bb, N, N] for some b >= 0. The first b indices index a batch member. For every batch index (i1,...,ib), A[i1,...,ib, : :] is an N x N matrix. This matrix A is not materialized, but for purposes of broadcasting this shape will be relevant.

LinearOperatorDiag is initialized with a (batch) vector.

# Create a 2 x 2 diagonal linear operator.  
diag = [1., -1.]  
operator = LinearOperatorDiag(diag)  
  
operator.to\_dense()  
==> [[1.,  0.]  
     [0., -1.]]  
  
operator.shape  
==> [2, 2]  
  
operator.log\_abs\_determinant()  
==> scalar Tensor  
  
x = ... Shape [2, 4] Tensor  
operator.matmul(x)  
==> Shape [2, 4] Tensor  
  
# Create a [2, 3] batch of 4 x 4 linear operators.  
diag = tf.random.normal(shape=[2, 3, 4])  
operator = LinearOperatorDiag(diag)  
  
# Create a shape [2, 1, 4, 2] vector.  Note that this shape is compatible  
# since the batch dimensions, [2, 1], are broadcast to  
# operator.batch\_shape = [2, 3].  
y = tf.random.normal(shape=[2, 1, 4, 2])  
x = operator.solve(y)  
==> operator.matmul(x) = y

#### Shape compatibility

This operator acts on [batch] matrix with compatible shape. x is a batch matrix with compatible shape for matmul and solve if

operator.shape = [B1,...,Bb] + [N, N],  with b >= 0  
x.shape =   [C1,...,Cc] + [N, R],  
and [C1,...,Cc] broadcasts with [B1,...,Bb] to [D1,...,Dd]

#### Performance

Suppose operator is a LinearOperatorDiag of shape [N, N], and x.shape = [N, R]. Then

* operator.matmul(x) involves N \* R multiplications.
* operator.solve(x) involves N divisions and N \* R multiplications.
* operator.determinant() involves a size N reduce\_prod.

If instead operator and x have shape [B1,...,Bb, N, N] and [B1,...,Bb, N, R], every operation increases in complexity by B1\*...\*Bb.

#### Matrix property hints

This LinearOperator is initialized with boolean flags of the form is\_X, for X = non\_singular, self\_adjoint, positive\_definite, square. These have the following meaning:

* If is\_X == True, callers should expect the operator to have the property X. This is a promise that should be fulfilled, but is not a runtime assert. For example, finite floating point precision may result in these promises being violated.
* If is\_X == False, callers should expect the operator to not have X.
* If is\_X == None (the default), callers should have no expectation either way.

## \_\_init\_\_

\_\_init\_\_(  
    diag,  
    is\_non\_singular=None,  
    is\_self\_adjoint=None,  
    is\_positive\_definite=None,  
    is\_square=None,  
    name='LinearOperatorDiag'  
)

Initialize a LinearOperatorDiag.

#### Args:

* **diag**: Shape [B1,...,Bb, N] Tensor with b >= 0 N >= 0. The diagonal of the operator. Allowed dtypes: float16, float32, float64, complex64, complex128.
* **is\_non\_singular**: Expect that this operator is non-singular.
* **is\_self\_adjoint**: Expect that this operator is equal to its hermitian transpose. If diag.dtypeis real, this is auto-set to True.
* **is\_positive\_definite**: Expect that this operator is positive definite, meaning the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive-definite. See: https://en.wikipedia.org/wiki/Positive-definite\_matrix#Extension\_for\_non-symmetric\_matrices
* **is\_square**: Expect that this operator acts like square [batch] matrices.
* **name**: A name for this LinearOperator.

#### Raises:

* **TypeError**: If diag.dtype is not an allowed type.
* **ValueError**: If diag.dtype is real, and is\_self\_adjoint is not True.

## Properties

### H

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### batch\_shape

TensorShape of batch dimensions of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb]), equivalent to A.get\_shape()[:-2]

#### Returns:

TensorShape, statically determined, may be undefined.

### diag

### domain\_dimension

Dimension (in the sense of vector spaces) of the domain of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Returns:

Dimension object.

### dtype

The DType of Tensors handled by this LinearOperator.

### graph\_parents

List of graph dependencies of this LinearOperator.

### is\_non\_singular

### is\_positive\_definite

### is\_self\_adjoint

### is\_square

Return True/False depending on if this operator is square.

### name

Name prepended to all ops created by this LinearOperator.

### range\_dimension

Dimension (in the sense of vector spaces) of the range of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Returns:

Dimension object.

### shape

TensorShape of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb, M, N]), equivalent to A.get\_shape().

#### Returns:

TensorShape, statically determined, may be undefined.

### tensor\_rank

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

Python integer, or None if the tensor rank is undefined.

## Methods

### add\_to\_tensor

add\_to\_tensor(  
    x,  
    name='add\_to\_tensor'  
)

Add matrix represented by this operator to x. Equivalent to A + x.

#### Args:

* **x**: Tensor with same dtype and shape broadcastable to self.shape.
* **name**: A name to give this Op.

#### Returns:

A Tensor with broadcast shape and same dtype as self.

### adjoint

adjoint(name='adjoint')

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### assert\_non\_singular

assert\_non\_singular(name='assert\_non\_singular')

Returns an Op that asserts this operator is non singular.

This operator is considered non-singular if

ConditionNumber < max{100, range\_dimension, domain\_dimension} \* eps,  
eps := np.finfo(self.dtype.as\_numpy\_dtype).eps

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is singular.

### assert\_positive\_definite

assert\_positive\_definite(name='assert\_positive\_definite')

Returns an Op that asserts this operator is positive definite.

Here, positive definite means that the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive definite.

#### Args:

* **name**: A name to give this Op.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not positive definite.

### assert\_self\_adjoint

assert\_self\_adjoint(name='assert\_self\_adjoint')

Returns an Op that asserts this operator is self-adjoint.

Here we check that this operator is exactly equal to its hermitian transpose.

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not self-adjoint.

### batch\_shape\_tensor

batch\_shape\_tensor(name='batch\_shape\_tensor')

Shape of batch dimensions of this operator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb].

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### cholesky

cholesky(name='cholesky')

Returns a Cholesky factor as a LinearOperator.

Given A representing this LinearOperator, if A is positive definite self-adjoint, return L, where A = L L^T, i.e. the cholesky decomposition.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the lower triangular matrix in the Cholesky decomposition.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be positive definite and self adjoint.

### determinant

determinant(name='det')

Determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### diag\_part

diag\_part(name='diag\_part')

Efficiently get the [batch] diagonal part of this operator.

If this operator has shape [B1,...,Bb, M, N], this returns a Tensor diagonal, of shape [B1,...,Bb, min(M, N)], where diagonal[b1,...,bb, i] = self.to\_dense()[b1,...,bb, i, i].

my\_operator = LinearOperatorDiag([1., 2.])  
  
# Efficiently get the diagonal  
my\_operator.diag\_part()  
==> [1., 2.]  
  
# Equivalent, but inefficient method  
tf.linalg.diag\_part(my\_operator.to\_dense())  
==> [1., 2.]

#### Args:

* **name**: A name for this Op.

#### Returns:

* **diag\_part**: A Tensor of same dtype as self.

### domain\_dimension\_tensor

domain\_dimension\_tensor(name='domain\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the domain of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### inverse

inverse(name='inverse')

Returns the Inverse of this LinearOperator.

Given A representing this LinearOperator, return a LinearOperator representing A^-1.

#### Args:

* **name**: A name scope to use for ops added by this method.

#### Returns:

LinearOperator representing inverse of this matrix.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be non\_singular.

### log\_abs\_determinant

log\_abs\_determinant(name='log\_abs\_det')

Log absolute value of determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### matmul

matmul(  
    x,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='matmul'  
)

Transform [batch] matrix x with left multiplication: x --> Ax.

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
X = ... # shape [..., N, R], batch matrix, R > 0.  
  
Y = operator.matmul(X)  
Y.shape  
==> [..., M, R]  
  
Y[..., :, r] = sum\_j A[..., :, j] X[j, r]

#### Args:

* **x**: LinearOperator or Tensor with compatible shape and same dtype as self. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **adjoint\_arg**: Python bool. If True, compute A x^H where x^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name for this Op.

#### Returns:

A LinearOperator or Tensor with shape [..., M, R] and same dtype as self.

### matvec

matvec(  
    x,  
    adjoint=False,  
    name='matvec'  
)

Transform [batch] vector x with left multiplication: x --> Ax.

# Make an operator acting like batch matric A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
  
X = ... # shape [..., N], batch vector  
  
Y = operator.matvec(X)  
Y.shape  
==> [..., M]  
  
Y[..., :] = sum\_j A[..., :, j] X[..., j]

#### Args:

* **x**: Tensor with compatible shape and same dtype as self. x is treated as a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **name**: A name for this Op.

#### Returns:

A Tensor with shape [..., M] and same dtype as self.

### range\_dimension\_tensor

range\_dimension\_tensor(name='range\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the range of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### shape\_tensor

shape\_tensor(name='shape\_tensor')

Shape of this LinearOperator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb, M, N], equivalent to tf.shape(A).

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### solve

solve(  
    rhs,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='solve'  
)

Solve (exact or approx) R (batch) systems of equations: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve R > 0 linear systems for every member of the batch.  
RHS = ... # shape [..., M, R]  
  
X = operator.solve(RHS)  
# X[..., :, r] is the solution to the r'th linear system  
# sum\_j A[..., :, j] X[..., j, r] = RHS[..., :, r]  
  
operator.matmul(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator and compatible shape. rhs is treated like a [batch] matrix meaning for every set of leading dimensions, the last two dimensions defines a matrix. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **adjoint\_arg**: Python bool. If True, solve A X = rhs^H where rhs^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N, R] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### solvevec

solvevec(  
    rhs,  
    adjoint=False,  
    name='solve'  
)

Solve single equation with best effort: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve one linear system for every member of the batch.  
RHS = ... # shape [..., M]  
  
X = operator.solvevec(RHS)  
# X is the solution to the linear system  
# sum\_j A[..., :, j] X[..., j] = RHS[..., :]  
  
operator.matvec(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator. rhs is treated like a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility regarding batch dimensions.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### tensor\_rank\_tensor

tensor\_rank\_tensor(name='tensor\_rank\_tensor')

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor, determined at runtime.

### to\_dense

to\_dense(name='to\_dense')

Return a dense (batch) matrix representing this operator.

### trace

trace(name='trace')

Trace of the linear operator, equal to sum of self.diag\_part().

If the operator is square, this is also the sum of the eigenvalues.

#### Args:

* **name**: A name for this Op.

#### Returns:

Shape [B1,...,Bb] Tensor of same dtype as self.

# tf.linalg.LinearOperatorFullMatrix

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorFullMatrix#top_of_page)
* [Class LinearOperatorFullMatrix](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorFullMatrix#class_linearoperatorfullmatrix)
  + [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorFullMatrix#aliases)
* [\_\_init\_\_](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorFullMatrix#__init__)
* [Properties](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorFullMatrix#properties)

## Class LinearOperatorFullMatrix

LinearOperator that wraps a [batch] matrix.

Inherits From: [LinearOperator](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperator)

### Aliases:

* Class tf.compat.v1.linalg.LinearOperatorFullMatrix
* Class tf.compat.v2.linalg.LinearOperatorFullMatrix
* Class tf.linalg.LinearOperatorFullMatrix

Defined in [python/ops/linalg/linear\_operator\_full\_matrix.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg/linear_operator_full_matrix.py).

This operator wraps a [batch] matrix A (which is a Tensor) with shape [B1,...,Bb, M, N] for some b >= 0. The first b indices index a batch member. For every batch index (i1,...,ib), A[i1,...,ib, : :] is an M x N matrix.

# Create a 2 x 2 linear operator.  
matrix = [[1., 2.], [3., 4.]]  
operator = LinearOperatorFullMatrix(matrix)  
  
operator.to\_dense()  
==> [[1., 2.]  
     [3., 4.]]  
  
operator.shape  
==> [2, 2]  
  
operator.log\_abs\_determinant()  
==> scalar Tensor  
  
x = ... Shape [2, 4] Tensor  
operator.matmul(x)  
==> Shape [2, 4] Tensor  
  
# Create a [2, 3] batch of 4 x 4 linear operators.  
matrix = tf.random.normal(shape=[2, 3, 4, 4])  
operator = LinearOperatorFullMatrix(matrix)

#### Shape compatibility

This operator acts on [batch] matrix with compatible shape. x is a batch matrix with compatible shape for matmul and solve if

operator.shape = [B1,...,Bb] + [M, N],  with b >= 0  
x.shape =        [B1,...,Bb] + [N, R],  with R >= 0.

#### Performance

LinearOperatorFullMatrix has exactly the same performance as would be achieved by using standard TensorFlow matrix ops. Intelligent choices are made based on the following initialization hints.

* If dtype is real, and is\_self\_adjoint and is\_positive\_definite, a Cholesky factorization is used for the determinant and solve.

In all cases, suppose operator is a LinearOperatorFullMatrix of shape [M, N], and x.shape = [N, R]. Then

* operator.matmul(x) is O(M \* N \* R).
* If M=N, operator.solve(x) is O(N^3 \* R).
* If M=N, operator.determinant() is O(N^3).

If instead operator and x have shape [B1,...,Bb, M, N] and [B1,...,Bb, N, R], every operation increases in complexity by B1\*...\*Bb.

#### Matrix property hints

This LinearOperator is initialized with boolean flags of the form is\_X, for X = non\_singular, self\_adjoint, positive\_definite, square. These have the following meaning:

* If is\_X == True, callers should expect the operator to have the property X. This is a promise that should be fulfilled, but is not a runtime assert. For example, finite floating point precision may result in these promises being violated.
* If is\_X == False, callers should expect the operator to not have X.
* If is\_X == None (the default), callers should have no expectation either way.

## \_\_init\_\_

\_\_init\_\_(  
    matrix,  
    is\_non\_singular=None,  
    is\_self\_adjoint=None,  
    is\_positive\_definite=None,  
    is\_square=None,  
    name='LinearOperatorFullMatrix'  
)

Initialize a LinearOperatorFullMatrix.

#### Args:

* **matrix**: Shape [B1,...,Bb, M, N] with b >= 0, M, N >= 0. Allowed dtypes: float16, float32, float64, complex64, complex128.
* **is\_non\_singular**: Expect that this operator is non-singular.
* **is\_self\_adjoint**: Expect that this operator is equal to its hermitian transpose.
* **is\_positive\_definite**: Expect that this operator is positive definite, meaning the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive-definite. See: https://en.wikipedia.org/wiki/Positive-definite\_matrix#Extension\_for\_non-symmetric\_matrices
* **is\_square**: Expect that this operator acts like square [batch] matrices.
* **name**: A name for this LinearOperator.

#### Raises:

* **TypeError**: If diag.dtype is not an allowed type.

## Properties

### H

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### batch\_shape

TensorShape of batch dimensions of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb]), equivalent to A.get\_shape()[:-2]

#### Returns:

TensorShape, statically determined, may be undefined.

### domain\_dimension

Dimension (in the sense of vector spaces) of the domain of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Returns:

Dimension object.

### dtype

The DType of Tensors handled by this LinearOperator.

### graph\_parents

List of graph dependencies of this LinearOperator.

### is\_non\_singular

### is\_positive\_definite

### is\_self\_adjoint

### is\_square

Return True/False depending on if this operator is square.

### name

Name prepended to all ops created by this LinearOperator.

### range\_dimension

Dimension (in the sense of vector spaces) of the range of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Returns:

Dimension object.

### shape

TensorShape of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb, M, N]), equivalent to A.get\_shape().

#### Returns:

TensorShape, statically determined, may be undefined.

### tensor\_rank

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

Python integer, or None if the tensor rank is undefined.

## Methods

### add\_to\_tensor

add\_to\_tensor(  
    x,  
    name='add\_to\_tensor'  
)

Add matrix represented by this operator to x. Equivalent to A + x.

#### Args:

* **x**: Tensor with same dtype and shape broadcastable to self.shape.
* **name**: A name to give this Op.

#### Returns:

A Tensor with broadcast shape and same dtype as self.

### adjoint

adjoint(name='adjoint')

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### assert\_non\_singular

assert\_non\_singular(name='assert\_non\_singular')

Returns an Op that asserts this operator is non singular.

This operator is considered non-singular if

ConditionNumber < max{100, range\_dimension, domain\_dimension} \* eps,  
eps := np.finfo(self.dtype.as\_numpy\_dtype).eps

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is singular.

### assert\_positive\_definite

assert\_positive\_definite(name='assert\_positive\_definite')

Returns an Op that asserts this operator is positive definite.

Here, positive definite means that the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive definite.

#### Args:

* **name**: A name to give this Op.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not positive definite.

### assert\_self\_adjoint

assert\_self\_adjoint(name='assert\_self\_adjoint')

Returns an Op that asserts this operator is self-adjoint.

Here we check that this operator is exactly equal to its hermitian transpose.

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not self-adjoint.

### batch\_shape\_tensor

batch\_shape\_tensor(name='batch\_shape\_tensor')

Shape of batch dimensions of this operator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb].

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### cholesky

cholesky(name='cholesky')

Returns a Cholesky factor as a LinearOperator.

Given A representing this LinearOperator, if A is positive definite self-adjoint, return L, where A = L L^T, i.e. the cholesky decomposition.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the lower triangular matrix in the Cholesky decomposition.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be positive definite and self adjoint.

### determinant

determinant(name='det')

Determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### diag\_part

diag\_part(name='diag\_part')

Efficiently get the [batch] diagonal part of this operator.

If this operator has shape [B1,...,Bb, M, N], this returns a Tensor diagonal, of shape [B1,...,Bb, min(M, N)], where diagonal[b1,...,bb, i] = self.to\_dense()[b1,...,bb, i, i].

my\_operator = LinearOperatorDiag([1., 2.])  
  
# Efficiently get the diagonal  
my\_operator.diag\_part()  
==> [1., 2.]  
  
# Equivalent, but inefficient method  
tf.linalg.diag\_part(my\_operator.to\_dense())  
==> [1., 2.]

#### Args:

* **name**: A name for this Op.

#### Returns:

* **diag\_part**: A Tensor of same dtype as self.

### domain\_dimension\_tensor

domain\_dimension\_tensor(name='domain\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the domain of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### inverse

inverse(name='inverse')

Returns the Inverse of this LinearOperator.

Given A representing this LinearOperator, return a LinearOperator representing A^-1.

#### Args:

* **name**: A name scope to use for ops added by this method.

#### Returns:

LinearOperator representing inverse of this matrix.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be non\_singular.

### log\_abs\_determinant

log\_abs\_determinant(name='log\_abs\_det')

Log absolute value of determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### matmul

matmul(  
    x,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='matmul'  
)

Transform [batch] matrix x with left multiplication: x --> Ax.

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
X = ... # shape [..., N, R], batch matrix, R > 0.  
  
Y = operator.matmul(X)  
Y.shape  
==> [..., M, R]  
  
Y[..., :, r] = sum\_j A[..., :, j] X[j, r]

#### Args:

* **x**: LinearOperator or Tensor with compatible shape and same dtype as self. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **adjoint\_arg**: Python bool. If True, compute A x^H where x^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name for this Op.

#### Returns:

A LinearOperator or Tensor with shape [..., M, R] and same dtype as self.

### matvec

matvec(  
    x,  
    adjoint=False,  
    name='matvec'  
)

Transform [batch] vector x with left multiplication: x --> Ax.

# Make an operator acting like batch matric A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
  
X = ... # shape [..., N], batch vector  
  
Y = operator.matvec(X)  
Y.shape  
==> [..., M]  
  
Y[..., :] = sum\_j A[..., :, j] X[..., j]

#### Args:

* **x**: Tensor with compatible shape and same dtype as self. x is treated as a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **name**: A name for this Op.

#### Returns:

A Tensor with shape [..., M] and same dtype as self.

### range\_dimension\_tensor

range\_dimension\_tensor(name='range\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the range of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### shape\_tensor

shape\_tensor(name='shape\_tensor')

Shape of this LinearOperator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb, M, N], equivalent to tf.shape(A).

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### solve

solve(  
    rhs,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='solve'  
)

Solve (exact or approx) R (batch) systems of equations: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve R > 0 linear systems for every member of the batch.  
RHS = ... # shape [..., M, R]  
  
X = operator.solve(RHS)  
# X[..., :, r] is the solution to the r'th linear system  
# sum\_j A[..., :, j] X[..., j, r] = RHS[..., :, r]  
  
operator.matmul(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator and compatible shape. rhs is treated like a [batch] matrix meaning for every set of leading dimensions, the last two dimensions defines a matrix. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **adjoint\_arg**: Python bool. If True, solve A X = rhs^H where rhs^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N, R] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### solvevec

solvevec(  
    rhs,  
    adjoint=False,  
    name='solve'  
)

Solve single equation with best effort: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve one linear system for every member of the batch.  
RHS = ... # shape [..., M]  
  
X = operator.solvevec(RHS)  
# X is the solution to the linear system  
# sum\_j A[..., :, j] X[..., j] = RHS[..., :]  
  
operator.matvec(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator. rhs is treated like a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility regarding batch dimensions.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### tensor\_rank\_tensor

tensor\_rank\_tensor(name='tensor\_rank\_tensor')

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor, determined at runtime.

### to\_dense

to\_dense(name='to\_dense')

Return a dense (batch) matrix representing this operator.

### trace

trace(name='trace')

Trace of the linear operator, equal to sum of self.diag\_part().

If the operator is square, this is also the sum of the eigenvalues.

#### Args:

* **name**: A name for this Op.

#### Returns:

Shape [B1,...,Bb] Tensor of same dtype as self.

# tf.linalg.LinearOperatorHouseholder

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorHouseholder#top_of_page)
* [Class LinearOperatorHouseholder](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorHouseholder#class_linearoperatorhouseholder)
  + [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorHouseholder#aliases)
* [Properties](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorHouseholder#properties)
  + [H](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorHouseholder#H)

## Class LinearOperatorHouseholder

LinearOperator acting like a [batch] of Householder transformations.

Inherits From: [LinearOperator](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperator)

### Aliases:

* Class tf.compat.v1.linalg.LinearOperatorHouseholder
* Class tf.compat.v2.linalg.LinearOperatorHouseholder
* Class tf.linalg.LinearOperatorHouseholder

Defined in [python/ops/linalg/linear\_operator\_householder.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg/linear_operator_householder.py).

This operator acts like a [batch] of householder reflections with shape [B1,...,Bb, N, N] for some b >= 0. The first b indices index a batch member. For every batch index (i1,...,ib), A[i1,...,ib, : :] is an N x N matrix. This matrix A is not materialized, but for purposes of broadcasting this shape will be relevant.

LinearOperatorHouseholder is initialized with a (batch) vector.

A Householder reflection, defined via a vector v, which reflects points in R^n about the hyperplane orthogonal to v and through the origin.

# Create a 2 x 2 householder transform.  
vec = [1 / np.sqrt(2), 1. / np.sqrt(2)]  
operator = LinearOperatorHouseholder(vec)  
  
operator.to\_dense()  
==> [[0.,  -1.]  
     [-1., -0.]]  
  
operator.shape  
==> [2, 2]  
  
operator.log\_abs\_determinant()  
==> scalar Tensor  
  
x = ... Shape [2, 4] Tensor  
operator.matmul(x)  
==> Shape [2, 4] Tensor  
  
#### Shape compatibility  
  
This operator acts on [batch] matrix with compatible shape.  
`x` is a batch matrix with compatible shape for `matmul` and `solve` if

operator.shape = [B1,...,Bb] + [N, N], with b >= 0 x.shape = [C1,...,Cc] + [N, R], and [C1,...,Cc] broadcasts with [B1,...,Bb] to [D1,...,Dd]

#### Matrix property hints  
  
This `LinearOperator` is initialized with boolean flags of the form `is\_X`,  
for `X = non\_singular, self\_adjoint, positive\_definite, square`.  
These have the following meaning:  
  
\* If `is\_X == True`, callers should expect the operator to have the  
  property `X`.  This is a promise that should be fulfilled, but is \*not\* a  
  runtime assert.  For example, finite floating point precision may result  
  in these promises being violated.  
\* If `is\_X == False`, callers should expect the operator to not have `X`.  
\* If `is\_X == None` (the default), callers should have no expectation either  
  way.  
  
<h2 id="\_\_init\_\_"><code>\_\_init\_\_</code></h2>  
  
``` python  
\_\_init\_\_(  
    reflection\_axis,  
    is\_non\_singular=None,  
    is\_self\_adjoint=None,  
    is\_positive\_definite=None,  
    is\_square=None,  
    name='LinearOperatorHouseholder'  
)

Initialize a LinearOperatorHouseholder.

#### Args:

* **reflection\_axis**: Shape [B1,...,Bb, N] Tensor with b >= 0 N >= 0. The vector defining the hyperplane to reflect about. Allowed dtypes: float16, float32, float64, complex64, complex128.
* **is\_non\_singular**: Expect that this operator is non-singular.
* **is\_self\_adjoint**: Expect that this operator is equal to its hermitian transpose. This is autoset to true
* **is\_positive\_definite**: Expect that this operator is positive definite, meaning the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive-definite. See: https://en.wikipedia.org/wiki/Positive-definite\_matrix#Extension\_for\_non-symmetric\_matrices This is autoset to false.
* **is\_square**: Expect that this operator acts like square [batch] matrices. This is autoset to true.
* **name**: A name for this LinearOperator.

#### Raises:

* **ValueError**: is\_self\_adjoint is not True, is\_positive\_definite is not False or is\_square is not True.

## Properties

### H

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### batch\_shape

TensorShape of batch dimensions of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb]), equivalent to A.get\_shape()[:-2]

#### Returns:

TensorShape, statically determined, may be undefined.

### domain\_dimension

Dimension (in the sense of vector spaces) of the domain of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Returns:

Dimension object.

### dtype

The DType of Tensors handled by this LinearOperator.

### graph\_parents

List of graph dependencies of this LinearOperator.

### is\_non\_singular

### is\_positive\_definite

### is\_self\_adjoint

### is\_square

Return True/False depending on if this operator is square.

### name

Name prepended to all ops created by this LinearOperator.

### range\_dimension

Dimension (in the sense of vector spaces) of the range of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Returns:

Dimension object.

### reflection\_axis

### shape

TensorShape of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb, M, N]), equivalent to A.get\_shape().

#### Returns:

TensorShape, statically determined, may be undefined.

### tensor\_rank

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

Python integer, or None if the tensor rank is undefined.

## Methods

### add\_to\_tensor

add\_to\_tensor(  
    x,  
    name='add\_to\_tensor'  
)

Add matrix represented by this operator to x. Equivalent to A + x.

#### Args:

* **x**: Tensor with same dtype and shape broadcastable to self.shape.
* **name**: A name to give this Op.

#### Returns:

A Tensor with broadcast shape and same dtype as self.

### adjoint

adjoint(name='adjoint')

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### assert\_non\_singular

assert\_non\_singular(name='assert\_non\_singular')

Returns an Op that asserts this operator is non singular.

This operator is considered non-singular if

ConditionNumber < max{100, range\_dimension, domain\_dimension} \* eps,  
eps := np.finfo(self.dtype.as\_numpy\_dtype).eps

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is singular.

### assert\_positive\_definite

assert\_positive\_definite(name='assert\_positive\_definite')

Returns an Op that asserts this operator is positive definite.

Here, positive definite means that the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive definite.

#### Args:

* **name**: A name to give this Op.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not positive definite.

### assert\_self\_adjoint

assert\_self\_adjoint(name='assert\_self\_adjoint')

Returns an Op that asserts this operator is self-adjoint.

Here we check that this operator is exactly equal to its hermitian transpose.

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not self-adjoint.

### batch\_shape\_tensor

batch\_shape\_tensor(name='batch\_shape\_tensor')

Shape of batch dimensions of this operator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb].

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### cholesky

cholesky(name='cholesky')

Returns a Cholesky factor as a LinearOperator.

Given A representing this LinearOperator, if A is positive definite self-adjoint, return L, where A = L L^T, i.e. the cholesky decomposition.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the lower triangular matrix in the Cholesky decomposition.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be positive definite and self adjoint.

### determinant

determinant(name='det')

Determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### diag\_part

diag\_part(name='diag\_part')

Efficiently get the [batch] diagonal part of this operator.

If this operator has shape [B1,...,Bb, M, N], this returns a Tensor diagonal, of shape [B1,...,Bb, min(M, N)], where diagonal[b1,...,bb, i] = self.to\_dense()[b1,...,bb, i, i].

my\_operator = LinearOperatorDiag([1., 2.])  
  
# Efficiently get the diagonal  
my\_operator.diag\_part()  
==> [1., 2.]  
  
# Equivalent, but inefficient method  
tf.linalg.diag\_part(my\_operator.to\_dense())  
==> [1., 2.]

#### Args:

* **name**: A name for this Op.

#### Returns:

* **diag\_part**: A Tensor of same dtype as self.

### domain\_dimension\_tensor

domain\_dimension\_tensor(name='domain\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the domain of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### inverse

inverse(name='inverse')

Returns the Inverse of this LinearOperator.

Given A representing this LinearOperator, return a LinearOperator representing A^-1.

#### Args:

* **name**: A name scope to use for ops added by this method.

#### Returns:

LinearOperator representing inverse of this matrix.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be non\_singular.

### log\_abs\_determinant

log\_abs\_determinant(name='log\_abs\_det')

Log absolute value of determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### matmul

matmul(  
    x,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='matmul'  
)

Transform [batch] matrix x with left multiplication: x --> Ax.

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
X = ... # shape [..., N, R], batch matrix, R > 0.  
  
Y = operator.matmul(X)  
Y.shape  
==> [..., M, R]  
  
Y[..., :, r] = sum\_j A[..., :, j] X[j, r]

#### Args:

* **x**: LinearOperator or Tensor with compatible shape and same dtype as self. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **adjoint\_arg**: Python bool. If True, compute A x^H where x^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name for this Op.

#### Returns:

A LinearOperator or Tensor with shape [..., M, R] and same dtype as self.

### matvec

matvec(  
    x,  
    adjoint=False,  
    name='matvec'  
)

Transform [batch] vector x with left multiplication: x --> Ax.

# Make an operator acting like batch matric A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
  
X = ... # shape [..., N], batch vector  
  
Y = operator.matvec(X)  
Y.shape  
==> [..., M]  
  
Y[..., :] = sum\_j A[..., :, j] X[..., j]

#### Args:

* **x**: Tensor with compatible shape and same dtype as self. x is treated as a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **name**: A name for this Op.

#### Returns:

A Tensor with shape [..., M] and same dtype as self.

### range\_dimension\_tensor

range\_dimension\_tensor(name='range\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the range of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### shape\_tensor

shape\_tensor(name='shape\_tensor')

Shape of this LinearOperator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb, M, N], equivalent to tf.shape(A).

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### solve

solve(  
    rhs,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='solve'  
)

Solve (exact or approx) R (batch) systems of equations: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve R > 0 linear systems for every member of the batch.  
RHS = ... # shape [..., M, R]  
  
X = operator.solve(RHS)  
# X[..., :, r] is the solution to the r'th linear system  
# sum\_j A[..., :, j] X[..., j, r] = RHS[..., :, r]  
  
operator.matmul(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator and compatible shape. rhs is treated like a [batch] matrix meaning for every set of leading dimensions, the last two dimensions defines a matrix. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **adjoint\_arg**: Python bool. If True, solve A X = rhs^H where rhs^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N, R] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### solvevec

solvevec(  
    rhs,  
    adjoint=False,  
    name='solve'  
)

Solve single equation with best effort: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve one linear system for every member of the batch.  
RHS = ... # shape [..., M]  
  
X = operator.solvevec(RHS)  
# X is the solution to the linear system  
# sum\_j A[..., :, j] X[..., j] = RHS[..., :]  
  
operator.matvec(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator. rhs is treated like a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility regarding batch dimensions.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### tensor\_rank\_tensor

tensor\_rank\_tensor(name='tensor\_rank\_tensor')

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor, determined at runtime.

### to\_dense

to\_dense(name='to\_dense')

Return a dense (batch) matrix representing this operator.

### trace

trace(name='trace')

Trace of the linear operator, equal to sum of self.diag\_part().

If the operator is square, this is also the sum of the eigenvalues.

#### Args:

* **name**: A name for this Op.

#### Returns:

Shape [B1,...,Bb] Tensor of same dtype as self.

# tf.linalg.LinearOperatorIdentity

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorIdentity#top_of_page)
* [Class LinearOperatorIdentity](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorIdentity#class_linearoperatoridentity)
  + [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorIdentity#aliases)
  + [Shape compatibility](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorIdentity#shape_compatibility)
  + [Performance](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorIdentity#performance)

## Class LinearOperatorIdentity

LinearOperator acting like a [batch] square identity matrix.

### Aliases:

* Class tf.compat.v1.linalg.LinearOperatorIdentity
* Class tf.compat.v2.linalg.LinearOperatorIdentity
* Class tf.linalg.LinearOperatorIdentity

Defined in [python/ops/linalg/linear\_operator\_identity.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg/linear_operator_identity.py).

This operator acts like a [batch] identity matrix A with shape [B1,...,Bb, N, N] for some b >= 0. The first b indices index a batch member. For every batch index (i1,...,ib), A[i1,...,ib, : :]is an N x N matrix. This matrix A is not materialized, but for purposes of broadcasting this shape will be relevant.

LinearOperatorIdentity is initialized with num\_rows, and optionally batch\_shape, and dtypearguments. If batch\_shape is None, this operator efficiently passes through all arguments. If batch\_shape is provided, broadcasting may occur, which will require making copies.

# Create a 2 x 2 identity matrix.  
operator = LinearOperatorIdentity(num\_rows=2, dtype=tf.float32)  
  
operator.to\_dense()  
==> [[1., 0.]  
     [0., 1.]]  
  
operator.shape  
==> [2, 2]  
  
operator.log\_abs\_determinant()  
==> 0.  
  
x = ... Shape [2, 4] Tensor  
operator.matmul(x)  
==> Shape [2, 4] Tensor, same as x.  
  
y = tf.random.normal(shape=[3, 2, 4])  
# Note that y.shape is compatible with operator.shape because operator.shape  
# is broadcast to [3, 2, 2].  
# This broadcast does NOT require copying data, since we can infer that y  
# will be passed through without changing shape.  We are always able to infer  
# this if the operator has no batch\_shape.  
x = operator.solve(y)  
==> Shape [3, 2, 4] Tensor, same as y.  
  
# Create a 2-batch of 2x2 identity matrices  
operator = LinearOperatorIdentity(num\_rows=2, batch\_shape=[2])  
operator.to\_dense()  
==> [[[1., 0.]  
      [0., 1.]],  
     [[1., 0.]  
      [0., 1.]]]  
  
# Here, even though the operator has a batch shape, the input is the same as  
# the output, so x can be passed through without a copy.  The operator is able  
# to detect that no broadcast is necessary because both x and the operator  
# have statically defined shape.  
x = ... Shape [2, 2, 3]  
operator.matmul(x)  
==> Shape [2, 2, 3] Tensor, same as x  
  
# Here the operator and x have different batch\_shape, and are broadcast.  
# This requires a copy, since the output is different size than the input.  
x = ... Shape [1, 2, 3]  
operator.matmul(x)  
==> Shape [2, 2, 3] Tensor, equal to [x, x]

### Shape compatibility

This operator acts on [batch] matrix with compatible shape. x is a batch matrix with compatible shape for matmul and solve if

operator.shape = [B1,...,Bb] + [N, N],  with b >= 0  
x.shape =   [C1,...,Cc] + [N, R],  
and [C1,...,Cc] broadcasts with [B1,...,Bb] to [D1,...,Dd]

### Performance

If batch\_shape initialization arg is None:

* operator.matmul(x) is O(1)
* operator.solve(x) is O(1)
* operator.determinant() is O(1)

If batch\_shape initialization arg is provided, and static checks cannot rule out the need to broadcast:

* operator.matmul(x) is O(D1\*...\*Dd\*N\*R)
* operator.solve(x) is O(D1\*...\*Dd\*N\*R)
* operator.determinant() is O(B1\*...\*Bb)

#### Matrix property hints

This LinearOperator is initialized with boolean flags of the form is\_X, for X = non\_singular, self\_adjoint, positive\_definite, square. These have the following meaning:

* If is\_X == True, callers should expect the operator to have the property X. This is a promise that should be fulfilled, but is not a runtime assert. For example, finite floating point precision may result in these promises being violated.
* If is\_X == False, callers should expect the operator to not have X.
* If is\_X == None (the default), callers should have no expectation either way.

## \_\_init\_\_

\_\_init\_\_(  
    num\_rows,  
    batch\_shape=None,  
    dtype=None,  
    is\_non\_singular=True,  
    is\_self\_adjoint=True,  
    is\_positive\_definite=True,  
    is\_square=True,  
    assert\_proper\_shapes=False,  
    name='LinearOperatorIdentity'  
)

Initialize a LinearOperatorIdentity.

The LinearOperatorIdentity is initialized with arguments defining dtype and shape.

This operator is able to broadcast the leading (batch) dimensions, which sometimes requires copying data. If batch\_shape is None, the operator can take arguments of any batch shape without copying. See examples.

#### Args:

* **num\_rows**: Scalar non-negative integer Tensor. Number of rows in the corresponding identity matrix.
* **batch\_shape**: Optional 1-D integer Tensor. The shape of the leading dimensions. If None, this operator has no leading dimensions.
* **dtype**: Data type of the matrix that this operator represents.
* **is\_non\_singular**: Expect that this operator is non-singular.
* **is\_self\_adjoint**: Expect that this operator is equal to its hermitian transpose.
* **is\_positive\_definite**: Expect that this operator is positive definite, meaning the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive-definite. See: https://en.wikipedia.org/wiki/Positive-definite\_matrix#Extension\_for\_non-symmetric\_matrices
* **is\_square**: Expect that this operator acts like square [batch] matrices.
* **assert\_proper\_shapes**: Python bool. If False, only perform static checks that initialization and method arguments have proper shape. If True, and static checks are inconclusive, add asserts to the graph.
* **name**: A name for this LinearOperator

#### Raises:

* **ValueError**: If num\_rows is determined statically to be non-scalar, or negative.
* **ValueError**: If batch\_shape is determined statically to not be 1-D, or negative.
* **ValueError**: If any of the following is not True: {is\_self\_adjoint, is\_non\_singular, is\_positive\_definite}.

## Properties

### H

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### batch\_shape

TensorShape of batch dimensions of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb]), equivalent to A.get\_shape()[:-2]

#### Returns:

TensorShape, statically determined, may be undefined.

### domain\_dimension

Dimension (in the sense of vector spaces) of the domain of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Returns:

Dimension object.

### dtype

The DType of Tensors handled by this LinearOperator.

### graph\_parents

List of graph dependencies of this LinearOperator.

### is\_non\_singular

### is\_positive\_definite

### is\_self\_adjoint

### is\_square

Return True/False depending on if this operator is square.

### name

Name prepended to all ops created by this LinearOperator.

### range\_dimension

Dimension (in the sense of vector spaces) of the range of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Returns:

Dimension object.

### shape

TensorShape of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb, M, N]), equivalent to A.get\_shape().

#### Returns:

TensorShape, statically determined, may be undefined.

### tensor\_rank

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

Python integer, or None if the tensor rank is undefined.

## Methods

### add\_to\_tensor

add\_to\_tensor(  
    mat,  
    name='add\_to\_tensor'  
)

Add matrix represented by this operator to mat. Equiv to I + mat.

#### Args:

* **mat**: Tensor with same dtype and shape broadcastable to self.
* **name**: A name to give this Op.

#### Returns:

A Tensor with broadcast shape and same dtype as self.

### adjoint

adjoint(name='adjoint')

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### assert\_non\_singular

assert\_non\_singular(name='assert\_non\_singular')

Returns an Op that asserts this operator is non singular.

This operator is considered non-singular if

ConditionNumber < max{100, range\_dimension, domain\_dimension} \* eps,  
eps := np.finfo(self.dtype.as\_numpy\_dtype).eps

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is singular.

### assert\_positive\_definite

assert\_positive\_definite(name='assert\_positive\_definite')

Returns an Op that asserts this operator is positive definite.

Here, positive definite means that the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive definite.

#### Args:

* **name**: A name to give this Op.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not positive definite.

### assert\_self\_adjoint

assert\_self\_adjoint(name='assert\_self\_adjoint')

Returns an Op that asserts this operator is self-adjoint.

Here we check that this operator is exactly equal to its hermitian transpose.

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not self-adjoint.

### batch\_shape\_tensor

batch\_shape\_tensor(name='batch\_shape\_tensor')

Shape of batch dimensions of this operator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb].

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### cholesky

cholesky(name='cholesky')

Returns a Cholesky factor as a LinearOperator.

Given A representing this LinearOperator, if A is positive definite self-adjoint, return L, where A = L L^T, i.e. the cholesky decomposition.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the lower triangular matrix in the Cholesky decomposition.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be positive definite and self adjoint.

### determinant

determinant(name='det')

Determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### diag\_part

diag\_part(name='diag\_part')

Efficiently get the [batch] diagonal part of this operator.

If this operator has shape [B1,...,Bb, M, N], this returns a Tensor diagonal, of shape [B1,...,Bb, min(M, N)], where diagonal[b1,...,bb, i] = self.to\_dense()[b1,...,bb, i, i].

my\_operator = LinearOperatorDiag([1., 2.])  
  
# Efficiently get the diagonal  
my\_operator.diag\_part()  
==> [1., 2.]  
  
# Equivalent, but inefficient method  
tf.linalg.diag\_part(my\_operator.to\_dense())  
==> [1., 2.]

#### Args:

* **name**: A name for this Op.

#### Returns:

* **diag\_part**: A Tensor of same dtype as self.

### domain\_dimension\_tensor

domain\_dimension\_tensor(name='domain\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the domain of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### inverse

inverse(name='inverse')

Returns the Inverse of this LinearOperator.

Given A representing this LinearOperator, return a LinearOperator representing A^-1.

#### Args:

* **name**: A name scope to use for ops added by this method.

#### Returns:

LinearOperator representing inverse of this matrix.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be non\_singular.

### log\_abs\_determinant

log\_abs\_determinant(name='log\_abs\_det')

Log absolute value of determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### matmul

matmul(  
    x,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='matmul'  
)

Transform [batch] matrix x with left multiplication: x --> Ax.

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
X = ... # shape [..., N, R], batch matrix, R > 0.  
  
Y = operator.matmul(X)  
Y.shape  
==> [..., M, R]  
  
Y[..., :, r] = sum\_j A[..., :, j] X[j, r]

#### Args:

* **x**: LinearOperator or Tensor with compatible shape and same dtype as self. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **adjoint\_arg**: Python bool. If True, compute A x^H where x^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name for this Op.

#### Returns:

A LinearOperator or Tensor with shape [..., M, R] and same dtype as self.

### matvec

matvec(  
    x,  
    adjoint=False,  
    name='matvec'  
)

Transform [batch] vector x with left multiplication: x --> Ax.

# Make an operator acting like batch matric A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
  
X = ... # shape [..., N], batch vector  
  
Y = operator.matvec(X)  
Y.shape  
==> [..., M]  
  
Y[..., :] = sum\_j A[..., :, j] X[..., j]

#### Args:

* **x**: Tensor with compatible shape and same dtype as self. x is treated as a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **name**: A name for this Op.

#### Returns:

A Tensor with shape [..., M] and same dtype as self.

### range\_dimension\_tensor

range\_dimension\_tensor(name='range\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the range of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### shape\_tensor

shape\_tensor(name='shape\_tensor')

Shape of this LinearOperator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb, M, N], equivalent to tf.shape(A).

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### solve

solve(  
    rhs,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='solve'  
)

Solve (exact or approx) R (batch) systems of equations: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve R > 0 linear systems for every member of the batch.  
RHS = ... # shape [..., M, R]  
  
X = operator.solve(RHS)  
# X[..., :, r] is the solution to the r'th linear system  
# sum\_j A[..., :, j] X[..., j, r] = RHS[..., :, r]  
  
operator.matmul(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator and compatible shape. rhs is treated like a [batch] matrix meaning for every set of leading dimensions, the last two dimensions defines a matrix. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **adjoint\_arg**: Python bool. If True, solve A X = rhs^H where rhs^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N, R] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### solvevec

solvevec(  
    rhs,  
    adjoint=False,  
    name='solve'  
)

Solve single equation with best effort: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve one linear system for every member of the batch.  
RHS = ... # shape [..., M]  
  
X = operator.solvevec(RHS)  
# X is the solution to the linear system  
# sum\_j A[..., :, j] X[..., j] = RHS[..., :]  
  
operator.matvec(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator. rhs is treated like a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility regarding batch dimensions.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### tensor\_rank\_tensor

tensor\_rank\_tensor(name='tensor\_rank\_tensor')

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor, determined at runtime.

### to\_dense

to\_dense(name='to\_dense')

Return a dense (batch) matrix representing this operator.

### trace

trace(name='trace')

Trace of the linear operator, equal to sum of self.diag\_part().

If the operator is square, this is also the sum of the eigenvalues.

#### Args:

* **name**: A name for this Op.

#### Returns:

Shape [B1,...,Bb] Tensor of same dtype as self.

# tf.linalg.LinearOperatorInversion

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorInversion#top_of_page)
* [Class LinearOperatorInversion](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorInversion#class_linearoperatorinversion)
  + [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorInversion#aliases)
* [\_\_init\_\_](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorInversion#__init__)
* [Properties](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorInversion#properties)

## Class LinearOperatorInversion

LinearOperator representing the inverse of another operator.

Inherits From: [LinearOperator](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperator)

### Aliases:

* Class tf.compat.v1.linalg.LinearOperatorInversion
* Class tf.compat.v2.linalg.LinearOperatorInversion
* Class tf.linalg.LinearOperatorInversion

Defined in [python/ops/linalg/linear\_operator\_inversion.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg/linear_operator_inversion.py).

This operator represents the inverse of another operator.

# Create a 2 x 2 linear operator.  
operator = LinearOperatorFullMatrix([[1., 0.], [0., 2.]])  
operator\_inv = LinearOperatorInversion(operator)  
  
operator\_inv.to\_dense()  
==> [[1., 0.]  
     [0., 0.5]]  
  
operator\_inv.shape  
==> [2, 2]  
  
operator\_inv.log\_abs\_determinant()  
==> - log(2)  
  
x = ... Shape [2, 4] Tensor  
operator\_inv.matmul(x)  
==> Shape [2, 4] Tensor, equal to operator.solve(x)

#### Performance

The performance of LinearOperatorInversion depends on the underlying operators performance:solve and matmul are swapped, and determinant is inverted.

#### Matrix property hints

This LinearOperator is initialized with boolean flags of the form is\_X, for X = non\_singular, self\_adjoint, positive\_definite, square. These have the following meaning:

* If is\_X == True, callers should expect the operator to have the property X. This is a promise that should be fulfilled, but is not a runtime assert. For example, finite floating point precision may result in these promises being violated.
* If is\_X == False, callers should expect the operator to not have X.
* If is\_X == None (the default), callers should have no expectation either way.

## \_\_init\_\_

\_\_init\_\_(  
    operator,  
    is\_non\_singular=None,  
    is\_self\_adjoint=None,  
    is\_positive\_definite=None,  
    is\_square=None,  
    name=None  
)

Initialize a LinearOperatorInversion.

LinearOperatorInversion is initialized with an operator A. The solve and matmul methods are effectively swapped. E.g.

A = MyLinearOperator(...)  
B = LinearOperatorInversion(A)  
x = [....]  # a vector  
  
assert A.matvec(x) == B.solvevec(x)

#### Args:

* **operator**: LinearOperator object. If operator.is\_non\_singular == False, an exception is raised. We do allow operator.is\_non\_singular == None, in which case this operator will have is\_non\_singular == None. Similarly for is\_self\_adjoint and is\_positive\_definite.
* **is\_non\_singular**: Expect that this operator is non-singular.
* **is\_self\_adjoint**: Expect that this operator is equal to its hermitian transpose.
* **is\_positive\_definite**: Expect that this operator is positive definite, meaning the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive-definite. See: https://en.wikipedia.org/wiki/Positive-definite\_matrix#Extension\_for\_non-symmetric\_matrices
* **is\_square**: Expect that this operator acts like square [batch] matrices.
* **name**: A name for this LinearOperator. Default is operator.name + "\_inv".

#### Raises:

* **ValueError**: If operator.is\_non\_singular is False.

## Properties

### H

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### batch\_shape

TensorShape of batch dimensions of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb]), equivalent to A.get\_shape()[:-2]

#### Returns:

TensorShape, statically determined, may be undefined.

### domain\_dimension

Dimension (in the sense of vector spaces) of the domain of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Returns:

Dimension object.

### dtype

The DType of Tensors handled by this LinearOperator.

### graph\_parents

List of graph dependencies of this LinearOperator.

### is\_non\_singular

### is\_positive\_definite

### is\_self\_adjoint

### is\_square

Return True/False depending on if this operator is square.

### name

Name prepended to all ops created by this LinearOperator.

### operator

The operator before inversion.

### range\_dimension

Dimension (in the sense of vector spaces) of the range of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Returns:

Dimension object.

### shape

TensorShape of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb, M, N]), equivalent to A.get\_shape().

#### Returns:

TensorShape, statically determined, may be undefined.

### tensor\_rank

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

Python integer, or None if the tensor rank is undefined.

## Methods

### add\_to\_tensor

add\_to\_tensor(  
    x,  
    name='add\_to\_tensor'  
)

Add matrix represented by this operator to x. Equivalent to A + x.

#### Args:

* **x**: Tensor with same dtype and shape broadcastable to self.shape.
* **name**: A name to give this Op.

#### Returns:

A Tensor with broadcast shape and same dtype as self.

### adjoint

adjoint(name='adjoint')

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### assert\_non\_singular

assert\_non\_singular(name='assert\_non\_singular')

Returns an Op that asserts this operator is non singular.

This operator is considered non-singular if

ConditionNumber < max{100, range\_dimension, domain\_dimension} \* eps,  
eps := np.finfo(self.dtype.as\_numpy\_dtype).eps

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is singular.

### assert\_positive\_definite

assert\_positive\_definite(name='assert\_positive\_definite')

Returns an Op that asserts this operator is positive definite.

Here, positive definite means that the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive definite.

#### Args:

* **name**: A name to give this Op.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not positive definite.

### assert\_self\_adjoint

assert\_self\_adjoint(name='assert\_self\_adjoint')

Returns an Op that asserts this operator is self-adjoint.

Here we check that this operator is exactly equal to its hermitian transpose.

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not self-adjoint.

### batch\_shape\_tensor

batch\_shape\_tensor(name='batch\_shape\_tensor')

Shape of batch dimensions of this operator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb].

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### cholesky

cholesky(name='cholesky')

Returns a Cholesky factor as a LinearOperator.

Given A representing this LinearOperator, if A is positive definite self-adjoint, return L, where A = L L^T, i.e. the cholesky decomposition.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the lower triangular matrix in the Cholesky decomposition.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be positive definite and self adjoint.

### determinant

determinant(name='det')

Determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### diag\_part

diag\_part(name='diag\_part')

Efficiently get the [batch] diagonal part of this operator.

If this operator has shape [B1,...,Bb, M, N], this returns a Tensor diagonal, of shape [B1,...,Bb, min(M, N)], where diagonal[b1,...,bb, i] = self.to\_dense()[b1,...,bb, i, i].

my\_operator = LinearOperatorDiag([1., 2.])  
  
# Efficiently get the diagonal  
my\_operator.diag\_part()  
==> [1., 2.]  
  
# Equivalent, but inefficient method  
tf.linalg.diag\_part(my\_operator.to\_dense())  
==> [1., 2.]

#### Args:

* **name**: A name for this Op.

#### Returns:

* **diag\_part**: A Tensor of same dtype as self.

### domain\_dimension\_tensor

domain\_dimension\_tensor(name='domain\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the domain of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### inverse

inverse(name='inverse')

Returns the Inverse of this LinearOperator.

Given A representing this LinearOperator, return a LinearOperator representing A^-1.

#### Args:

* **name**: A name scope to use for ops added by this method.

#### Returns:

LinearOperator representing inverse of this matrix.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be non\_singular.

### log\_abs\_determinant

log\_abs\_determinant(name='log\_abs\_det')

Log absolute value of determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### matmul

matmul(  
    x,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='matmul'  
)

Transform [batch] matrix x with left multiplication: x --> Ax.

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
X = ... # shape [..., N, R], batch matrix, R > 0.  
  
Y = operator.matmul(X)  
Y.shape  
==> [..., M, R]  
  
Y[..., :, r] = sum\_j A[..., :, j] X[j, r]

#### Args:

* **x**: LinearOperator or Tensor with compatible shape and same dtype as self. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **adjoint\_arg**: Python bool. If True, compute A x^H where x^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name for this Op.

#### Returns:

A LinearOperator or Tensor with shape [..., M, R] and same dtype as self.

### matvec

matvec(  
    x,  
    adjoint=False,  
    name='matvec'  
)

Transform [batch] vector x with left multiplication: x --> Ax.

# Make an operator acting like batch matric A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
  
X = ... # shape [..., N], batch vector  
  
Y = operator.matvec(X)  
Y.shape  
==> [..., M]  
  
Y[..., :] = sum\_j A[..., :, j] X[..., j]

#### Args:

* **x**: Tensor with compatible shape and same dtype as self. x is treated as a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **name**: A name for this Op.

#### Returns:

A Tensor with shape [..., M] and same dtype as self.

### range\_dimension\_tensor

range\_dimension\_tensor(name='range\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the range of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### shape\_tensor

shape\_tensor(name='shape\_tensor')

Shape of this LinearOperator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb, M, N], equivalent to tf.shape(A).

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### solve

solve(  
    rhs,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='solve'  
)

Solve (exact or approx) R (batch) systems of equations: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve R > 0 linear systems for every member of the batch.  
RHS = ... # shape [..., M, R]  
  
X = operator.solve(RHS)  
# X[..., :, r] is the solution to the r'th linear system  
# sum\_j A[..., :, j] X[..., j, r] = RHS[..., :, r]  
  
operator.matmul(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator and compatible shape. rhs is treated like a [batch] matrix meaning for every set of leading dimensions, the last two dimensions defines a matrix. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **adjoint\_arg**: Python bool. If True, solve A X = rhs^H where rhs^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N, R] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### solvevec

solvevec(  
    rhs,  
    adjoint=False,  
    name='solve'  
)

Solve single equation with best effort: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve one linear system for every member of the batch.  
RHS = ... # shape [..., M]  
  
X = operator.solvevec(RHS)  
# X is the solution to the linear system  
# sum\_j A[..., :, j] X[..., j] = RHS[..., :]  
  
operator.matvec(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator. rhs is treated like a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility regarding batch dimensions.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### tensor\_rank\_tensor

tensor\_rank\_tensor(name='tensor\_rank\_tensor')

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor, determined at runtime.

### to\_dense

to\_dense(name='to\_dense')

Return a dense (batch) matrix representing this operator.

### trace

trace(name='trace')

Trace of the linear operator, equal to sum of self.diag\_part().

If the operator is square, this is also the sum of the eigenvalues.

#### Args:

* **name**: A name for this Op.

#### Returns:

Shape [B1,...,Bb] Tensor of same dtype as self.

# tf.linalg.LinearOperatorKronecker

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorKronecker#top_of_page)
* [Class LinearOperatorKronecker](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorKronecker#class_linearoperatorkronecker)
  + [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorKronecker#aliases)
* [\_\_init\_\_](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorKronecker#__init__)
* [Properties](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorKronecker#properties)

## Class LinearOperatorKronecker

Kronecker product between two LinearOperators.

Inherits From: [LinearOperator](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperator)

### Aliases:

* Class tf.compat.v1.linalg.LinearOperatorKronecker
* Class tf.compat.v2.linalg.LinearOperatorKronecker
* Class tf.linalg.LinearOperatorKronecker

Defined in [python/ops/linalg/linear\_operator\_kronecker.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg/linear_operator_kronecker.py).

This operator composes one or more linear operators [op1,...,opJ], building a new LinearOperator representing the Kronecker product: op1 x op2 x .. opJ (we omit parentheses as the Kronecker product is associative).

If opj has shape batch\_shape\_j + [M\_j, N\_j], then the composed operator will have shape equal to broadcast\_batch\_shape + [prod M\_j, prod N\_j], where the product is over all operators.

# Create a 4 x 4 linear operator composed of two 2 x 2 operators.  
operator\_1 = LinearOperatorFullMatrix([[1., 2.], [3., 4.]])  
operator\_2 = LinearOperatorFullMatrix([[1., 0.], [2., 1.]])  
operator = LinearOperatorKronecker([operator\_1, operator\_2])  
  
operator.to\_dense()  
==> [[1., 2., 0., 0.],  
     [3., 4., 0., 0.],  
     [2., 4., 1., 2.],  
     [6., 8., 3., 4.]]  
  
operator.shape  
==> [4, 4]  
  
operator.log\_abs\_determinant()  
==> scalar Tensor  
  
x = ... Shape [4, 2] Tensor  
operator.matmul(x)  
==> Shape [4, 2] Tensor  
  
# Create a [2, 3] batch of 4 x 5 linear operators.  
matrix\_45 = tf.random.normal(shape=[2, 3, 4, 5])  
operator\_45 = LinearOperatorFullMatrix(matrix)  
  
# Create a [2, 3] batch of 5 x 6 linear operators.  
matrix\_56 = tf.random.normal(shape=[2, 3, 5, 6])  
operator\_56 = LinearOperatorFullMatrix(matrix\_56)  
  
# Compose to create a [2, 3] batch of 20 x 30 operators.  
operator\_large = LinearOperatorKronecker([operator\_45, operator\_56])  
  
# Create a shape [2, 3, 20, 2] vector.  
x = tf.random.normal(shape=[2, 3, 6, 2])  
operator\_large.matmul(x)  
==> Shape [2, 3, 30, 2] Tensor

#### Performance

The performance of LinearOperatorKronecker on any operation is equal to the sum of the individual operators' operations.

#### Matrix property hints

This LinearOperator is initialized with boolean flags of the form is\_X, for X = non\_singular, self\_adjoint, positive\_definite, square. These have the following meaning:

* If is\_X == True, callers should expect the operator to have the property X. This is a promise that should be fulfilled, but is not a runtime assert. For example, finite floating point precision may result in these promises being violated.
* If is\_X == False, callers should expect the operator to not have X.
* If is\_X == None (the default), callers should have no expectation either way.

## \_\_init\_\_

\_\_init\_\_(  
    operators,  
    is\_non\_singular=None,  
    is\_self\_adjoint=None,  
    is\_positive\_definite=None,  
    is\_square=None,  
    name=None  
)

Initialize a LinearOperatorKronecker.

LinearOperatorKronecker is initialized with a list of operators [op\_1,...,op\_J].

#### Args:

* **operators**: Iterable of LinearOperator objects, each with the same dtype and composable shape, representing the Kronecker factors.
* **is\_non\_singular**: Expect that this operator is non-singular.
* **is\_self\_adjoint**: Expect that this operator is equal to its hermitian transpose.
* **is\_positive\_definite**: Expect that this operator is positive definite, meaning the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive-definite. See: https://en.wikipedia.org/wiki/Positive-definite\_matrix  
  #Extension\_for\_non\_symmetric\_matrices
* **is\_square**: Expect that this operator acts like square [batch] matrices.
* **name**: A name for this LinearOperator. Default is the individual operators names joined with \_x\_.

#### Raises:

* **TypeError**: If all operators do not have the same dtype.
* **ValueError**: If operators is empty.

## Properties

### H

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### batch\_shape

TensorShape of batch dimensions of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb]), equivalent to A.get\_shape()[:-2]

#### Returns:

TensorShape, statically determined, may be undefined.

### domain\_dimension

Dimension (in the sense of vector spaces) of the domain of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Returns:

Dimension object.

### dtype

The DType of Tensors handled by this LinearOperator.

### graph\_parents

List of graph dependencies of this LinearOperator.

### is\_non\_singular

### is\_positive\_definite

### is\_self\_adjoint

### is\_square

Return True/False depending on if this operator is square.

### name

Name prepended to all ops created by this LinearOperator.

### operators

### range\_dimension

Dimension (in the sense of vector spaces) of the range of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Returns:

Dimension object.

### shape

TensorShape of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb, M, N]), equivalent to A.get\_shape().

#### Returns:

TensorShape, statically determined, may be undefined.

### tensor\_rank

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

Python integer, or None if the tensor rank is undefined.

## Methods

### add\_to\_tensor

add\_to\_tensor(  
    x,  
    name='add\_to\_tensor'  
)

Add matrix represented by this operator to x. Equivalent to A + x.

#### Args:

* **x**: Tensor with same dtype and shape broadcastable to self.shape.
* **name**: A name to give this Op.

#### Returns:

A Tensor with broadcast shape and same dtype as self.

### adjoint

adjoint(name='adjoint')

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### assert\_non\_singular

assert\_non\_singular(name='assert\_non\_singular')

Returns an Op that asserts this operator is non singular.

This operator is considered non-singular if

ConditionNumber < max{100, range\_dimension, domain\_dimension} \* eps,  
eps := np.finfo(self.dtype.as\_numpy\_dtype).eps

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is singular.

### assert\_positive\_definite

assert\_positive\_definite(name='assert\_positive\_definite')

Returns an Op that asserts this operator is positive definite.

Here, positive definite means that the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive definite.

#### Args:

* **name**: A name to give this Op.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not positive definite.

### assert\_self\_adjoint

assert\_self\_adjoint(name='assert\_self\_adjoint')

Returns an Op that asserts this operator is self-adjoint.

Here we check that this operator is exactly equal to its hermitian transpose.

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not self-adjoint.

### batch\_shape\_tensor

batch\_shape\_tensor(name='batch\_shape\_tensor')

Shape of batch dimensions of this operator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb].

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### cholesky

cholesky(name='cholesky')

Returns a Cholesky factor as a LinearOperator.

Given A representing this LinearOperator, if A is positive definite self-adjoint, return L, where A = L L^T, i.e. the cholesky decomposition.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the lower triangular matrix in the Cholesky decomposition.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be positive definite and self adjoint.

### determinant

determinant(name='det')

Determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### diag\_part

diag\_part(name='diag\_part')

Efficiently get the [batch] diagonal part of this operator.

If this operator has shape [B1,...,Bb, M, N], this returns a Tensor diagonal, of shape [B1,...,Bb, min(M, N)], where diagonal[b1,...,bb, i] = self.to\_dense()[b1,...,bb, i, i].

my\_operator = LinearOperatorDiag([1., 2.])  
  
# Efficiently get the diagonal  
my\_operator.diag\_part()  
==> [1., 2.]  
  
# Equivalent, but inefficient method  
tf.linalg.diag\_part(my\_operator.to\_dense())  
==> [1., 2.]

#### Args:

* **name**: A name for this Op.

#### Returns:

* **diag\_part**: A Tensor of same dtype as self.

### domain\_dimension\_tensor

domain\_dimension\_tensor(name='domain\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the domain of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### inverse

inverse(name='inverse')

Returns the Inverse of this LinearOperator.

Given A representing this LinearOperator, return a LinearOperator representing A^-1.

#### Args:

* **name**: A name scope to use for ops added by this method.

#### Returns:

LinearOperator representing inverse of this matrix.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be non\_singular.

### log\_abs\_determinant

log\_abs\_determinant(name='log\_abs\_det')

Log absolute value of determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### matmul

matmul(  
    x,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='matmul'  
)

Transform [batch] matrix x with left multiplication: x --> Ax.

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
X = ... # shape [..., N, R], batch matrix, R > 0.  
  
Y = operator.matmul(X)  
Y.shape  
==> [..., M, R]  
  
Y[..., :, r] = sum\_j A[..., :, j] X[j, r]

#### Args:

* **x**: LinearOperator or Tensor with compatible shape and same dtype as self. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **adjoint\_arg**: Python bool. If True, compute A x^H where x^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name for this Op.

#### Returns:

A LinearOperator or Tensor with shape [..., M, R] and same dtype as self.

### matvec

matvec(  
    x,  
    adjoint=False,  
    name='matvec'  
)

Transform [batch] vector x with left multiplication: x --> Ax.

# Make an operator acting like batch matric A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
  
X = ... # shape [..., N], batch vector  
  
Y = operator.matvec(X)  
Y.shape  
==> [..., M]  
  
Y[..., :] = sum\_j A[..., :, j] X[..., j]

#### Args:

* **x**: Tensor with compatible shape and same dtype as self. x is treated as a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **name**: A name for this Op.

#### Returns:

A Tensor with shape [..., M] and same dtype as self.

### range\_dimension\_tensor

range\_dimension\_tensor(name='range\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the range of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### shape\_tensor

shape\_tensor(name='shape\_tensor')

Shape of this LinearOperator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb, M, N], equivalent to tf.shape(A).

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### solve

solve(  
    rhs,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='solve'  
)

Solve (exact or approx) R (batch) systems of equations: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve R > 0 linear systems for every member of the batch.  
RHS = ... # shape [..., M, R]  
  
X = operator.solve(RHS)  
# X[..., :, r] is the solution to the r'th linear system  
# sum\_j A[..., :, j] X[..., j, r] = RHS[..., :, r]  
  
operator.matmul(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator and compatible shape. rhs is treated like a [batch] matrix meaning for every set of leading dimensions, the last two dimensions defines a matrix. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **adjoint\_arg**: Python bool. If True, solve A X = rhs^H where rhs^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N, R] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### solvevec

solvevec(  
    rhs,  
    adjoint=False,  
    name='solve'  
)

Solve single equation with best effort: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve one linear system for every member of the batch.  
RHS = ... # shape [..., M]  
  
X = operator.solvevec(RHS)  
# X is the solution to the linear system  
# sum\_j A[..., :, j] X[..., j] = RHS[..., :]  
  
operator.matvec(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator. rhs is treated like a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility regarding batch dimensions.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### tensor\_rank\_tensor

tensor\_rank\_tensor(name='tensor\_rank\_tensor')

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor, determined at runtime.

### to\_dense

to\_dense(name='to\_dense')

Return a dense (batch) matrix representing this operator.

### trace

trace(name='trace')

Trace of the linear operator, equal to sum of self.diag\_part().

If the operator is square, this is also the sum of the eigenvalues.

#### Args:

* **name**: A name for this Op.

#### Returns:

Shape [B1,...,Bb] Tensor of same dtype as self.

# tf.linalg.LinearOperatorLowerTriangular

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorLowerTriangular#top_of_page)
* [Class LinearOperatorLowerTriangular](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorLowerTriangular#class_linearoperatorlowertriangular)
  + [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorLowerTriangular#aliases)
* [\_\_init\_\_](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorLowerTriangular#__init__)
* [Properties](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorLowerTriangular#properties)

## Class LinearOperatorLowerTriangular

LinearOperator acting like a [batch] square lower triangular matrix.

Inherits From: [LinearOperator](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperator)

### Aliases:

* Class tf.compat.v1.linalg.LinearOperatorLowerTriangular
* Class tf.compat.v2.linalg.LinearOperatorLowerTriangular
* Class tf.linalg.LinearOperatorLowerTriangular

Defined in [python/ops/linalg/linear\_operator\_lower\_triangular.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg/linear_operator_lower_triangular.py).

This operator acts like a [batch] lower triangular matrix A with shape [B1,...,Bb, N, N] for some b >= 0. The first b indices index a batch member. For every batch index (i1,...,ib), A[i1,...,ib, : :] is an N x N matrix.

LinearOperatorLowerTriangular is initialized with a Tensor having dimensions [B1,...,Bb, N, N]. The upper triangle of the last two dimensions is ignored.

# Create a 2 x 2 lower-triangular linear operator.  
tril = [[1., 2.], [3., 4.]]  
operator = LinearOperatorLowerTriangular(tril)  
  
# The upper triangle is ignored.  
operator.to\_dense()  
==> [[1., 0.]  
     [3., 4.]]  
  
operator.shape  
==> [2, 2]  
  
operator.log\_abs\_determinant()  
==> scalar Tensor  
  
x = ... Shape [2, 4] Tensor  
operator.matmul(x)  
==> Shape [2, 4] Tensor  
  
# Create a [2, 3] batch of 4 x 4 linear operators.  
tril = tf.random.normal(shape=[2, 3, 4, 4])  
operator = LinearOperatorLowerTriangular(tril)

#### Shape compatibility

This operator acts on [batch] matrix with compatible shape. x is a batch matrix with compatible shape for matmul and solve if

operator.shape = [B1,...,Bb] + [N, N],  with b >= 0  
x.shape =        [B1,...,Bb] + [N, R],  with R >= 0.

#### Performance

Suppose operator is a LinearOperatorLowerTriangular of shape [N, N], and x.shape = [N, R]. Then

* operator.matmul(x) involves N^2 \* R multiplications.
* operator.solve(x) involves N \* R size N back-substitutions.
* operator.determinant() involves a size N reduce\_prod.

If instead operator and x have shape [B1,...,Bb, N, N] and [B1,...,Bb, N, R], every operation increases in complexity by B1\*...\*Bb.

#### Matrix property hints

This LinearOperator is initialized with boolean flags of the form is\_X, for X = non\_singular, self\_adjoint, positive\_definite, square. These have the following meaning:

* If is\_X == True, callers should expect the operator to have the property X. This is a promise that should be fulfilled, but is not a runtime assert. For example, finite floating point precision may result in these promises being violated.
* If is\_X == False, callers should expect the operator to not have X.
* If is\_X == None (the default), callers should have no expectation either way.

## \_\_init\_\_

\_\_init\_\_(  
    tril,  
    is\_non\_singular=None,  
    is\_self\_adjoint=None,  
    is\_positive\_definite=None,  
    is\_square=None,  
    name='LinearOperatorLowerTriangular'  
)

Initialize a LinearOperatorLowerTriangular.

#### Args:

* **tril**: Shape [B1,...,Bb, N, N] with b >= 0, N >= 0. The lower triangular part of trildefines this operator. The strictly upper triangle is ignored.
* **is\_non\_singular**: Expect that this operator is non-singular. This operator is non-singular if and only if its diagonal elements are all non-zero.
* **is\_self\_adjoint**: Expect that this operator is equal to its hermitian transpose. This operator is self-adjoint only if it is diagonal with real-valued diagonal entries. In this case it is advised to useLinearOperatorDiag.
* **is\_positive\_definite**: Expect that this operator is positive definite, meaning the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive-definite. See: https://en.wikipedia.org/wiki/Positive-definite\_matrix#Extension\_for\_non-symmetric\_matrices
* **is\_square**: Expect that this operator acts like square [batch] matrices.
* **name**: A name for this LinearOperator.

#### Raises:

* **ValueError**: If is\_square is False.

## Properties

### H

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### batch\_shape

TensorShape of batch dimensions of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb]), equivalent to A.get\_shape()[:-2]

#### Returns:

TensorShape, statically determined, may be undefined.

### domain\_dimension

Dimension (in the sense of vector spaces) of the domain of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Returns:

Dimension object.

### dtype

The DType of Tensors handled by this LinearOperator.

### graph\_parents

List of graph dependencies of this LinearOperator.

### is\_non\_singular

### is\_positive\_definite

### is\_self\_adjoint

### is\_square

Return True/False depending on if this operator is square.

### name

Name prepended to all ops created by this LinearOperator.

### range\_dimension

Dimension (in the sense of vector spaces) of the range of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Returns:

Dimension object.

### shape

TensorShape of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb, M, N]), equivalent to A.get\_shape().

#### Returns:

TensorShape, statically determined, may be undefined.

### tensor\_rank

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

Python integer, or None if the tensor rank is undefined.

## Methods

### add\_to\_tensor

add\_to\_tensor(  
    x,  
    name='add\_to\_tensor'  
)

Add matrix represented by this operator to x. Equivalent to A + x.

#### Args:

* **x**: Tensor with same dtype and shape broadcastable to self.shape.
* **name**: A name to give this Op.

#### Returns:

A Tensor with broadcast shape and same dtype as self.

### adjoint

adjoint(name='adjoint')

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### assert\_non\_singular

assert\_non\_singular(name='assert\_non\_singular')

Returns an Op that asserts this operator is non singular.

This operator is considered non-singular if

ConditionNumber < max{100, range\_dimension, domain\_dimension} \* eps,  
eps := np.finfo(self.dtype.as\_numpy\_dtype).eps

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is singular.

### assert\_positive\_definite

assert\_positive\_definite(name='assert\_positive\_definite')

Returns an Op that asserts this operator is positive definite.

Here, positive definite means that the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive definite.

#### Args:

* **name**: A name to give this Op.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not positive definite.

### assert\_self\_adjoint

assert\_self\_adjoint(name='assert\_self\_adjoint')

Returns an Op that asserts this operator is self-adjoint.

Here we check that this operator is exactly equal to its hermitian transpose.

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not self-adjoint.

### batch\_shape\_tensor

batch\_shape\_tensor(name='batch\_shape\_tensor')

Shape of batch dimensions of this operator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb].

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### cholesky

cholesky(name='cholesky')

Returns a Cholesky factor as a LinearOperator.

Given A representing this LinearOperator, if A is positive definite self-adjoint, return L, where A = L L^T, i.e. the cholesky decomposition.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the lower triangular matrix in the Cholesky decomposition.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be positive definite and self adjoint.

### determinant

determinant(name='det')

Determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### diag\_part

diag\_part(name='diag\_part')

Efficiently get the [batch] diagonal part of this operator.

If this operator has shape [B1,...,Bb, M, N], this returns a Tensor diagonal, of shape [B1,...,Bb, min(M, N)], where diagonal[b1,...,bb, i] = self.to\_dense()[b1,...,bb, i, i].

my\_operator = LinearOperatorDiag([1., 2.])  
  
# Efficiently get the diagonal  
my\_operator.diag\_part()  
==> [1., 2.]  
  
# Equivalent, but inefficient method  
tf.linalg.diag\_part(my\_operator.to\_dense())  
==> [1., 2.]

#### Args:

* **name**: A name for this Op.

#### Returns:

* **diag\_part**: A Tensor of same dtype as self.

### domain\_dimension\_tensor

domain\_dimension\_tensor(name='domain\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the domain of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### inverse

inverse(name='inverse')

Returns the Inverse of this LinearOperator.

Given A representing this LinearOperator, return a LinearOperator representing A^-1.

#### Args:

* **name**: A name scope to use for ops added by this method.

#### Returns:

LinearOperator representing inverse of this matrix.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be non\_singular.

### log\_abs\_determinant

log\_abs\_determinant(name='log\_abs\_det')

Log absolute value of determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### matmul

matmul(  
    x,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='matmul'  
)

Transform [batch] matrix x with left multiplication: x --> Ax.

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
X = ... # shape [..., N, R], batch matrix, R > 0.  
  
Y = operator.matmul(X)  
Y.shape  
==> [..., M, R]  
  
Y[..., :, r] = sum\_j A[..., :, j] X[j, r]

#### Args:

* **x**: LinearOperator or Tensor with compatible shape and same dtype as self. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **adjoint\_arg**: Python bool. If True, compute A x^H where x^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name for this Op.

#### Returns:

A LinearOperator or Tensor with shape [..., M, R] and same dtype as self.

### matvec

matvec(  
    x,  
    adjoint=False,  
    name='matvec'  
)

Transform [batch] vector x with left multiplication: x --> Ax.

# Make an operator acting like batch matric A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
  
X = ... # shape [..., N], batch vector  
  
Y = operator.matvec(X)  
Y.shape  
==> [..., M]  
  
Y[..., :] = sum\_j A[..., :, j] X[..., j]

#### Args:

* **x**: Tensor with compatible shape and same dtype as self. x is treated as a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **name**: A name for this Op.

#### Returns:

A Tensor with shape [..., M] and same dtype as self.

### range\_dimension\_tensor

range\_dimension\_tensor(name='range\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the range of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### shape\_tensor

shape\_tensor(name='shape\_tensor')

Shape of this LinearOperator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb, M, N], equivalent to tf.shape(A).

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### solve

solve(  
    rhs,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='solve'  
)

Solve (exact or approx) R (batch) systems of equations: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve R > 0 linear systems for every member of the batch.  
RHS = ... # shape [..., M, R]  
  
X = operator.solve(RHS)  
# X[..., :, r] is the solution to the r'th linear system  
# sum\_j A[..., :, j] X[..., j, r] = RHS[..., :, r]  
  
operator.matmul(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator and compatible shape. rhs is treated like a [batch] matrix meaning for every set of leading dimensions, the last two dimensions defines a matrix. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **adjoint\_arg**: Python bool. If True, solve A X = rhs^H where rhs^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N, R] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### solvevec

solvevec(  
    rhs,  
    adjoint=False,  
    name='solve'  
)

Solve single equation with best effort: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve one linear system for every member of the batch.  
RHS = ... # shape [..., M]  
  
X = operator.solvevec(RHS)  
# X is the solution to the linear system  
# sum\_j A[..., :, j] X[..., j] = RHS[..., :]  
  
operator.matvec(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator. rhs is treated like a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility regarding batch dimensions.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### tensor\_rank\_tensor

tensor\_rank\_tensor(name='tensor\_rank\_tensor')

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor, determined at runtime.

### to\_dense

to\_dense(name='to\_dense')

Return a dense (batch) matrix representing this operator.

### trace

trace(name='trace')

Trace of the linear operator, equal to sum of self.diag\_part().

If the operator is square, this is also the sum of the eigenvalues.

#### Args:

* **name**: A name for this Op.

#### Returns:

Shape [B1,...,Bb] Tensor of same dtype as self.

# tf.linalg.LinearOperatorLowRankUpdate

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorLowRankUpdate#top_of_page)
* [Class LinearOperatorLowRankUpdate](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorLowRankUpdate#class_linearoperatorlowrankupdate)
  + [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorLowRankUpdate#aliases)
  + [Shape compatibility](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorLowRankUpdate#shape_compatibility)
  + [Performance](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorLowRankUpdate#performance)

## Class LinearOperatorLowRankUpdate

Perturb a LinearOperator with a rank K update.

Inherits From: [LinearOperator](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperator)

### Aliases:

* Class tf.compat.v1.linalg.LinearOperatorLowRankUpdate
* Class tf.compat.v2.linalg.LinearOperatorLowRankUpdate
* Class tf.linalg.LinearOperatorLowRankUpdate

Defined in [python/ops/linalg/linear\_operator\_low\_rank\_update.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg/linear_operator_low_rank_update.py).

This operator acts like a [batch] matrix A with shape [B1,...,Bb, M, N] for some b >= 0. The first b indices index a batch member. For every batch index (i1,...,ib), A[i1,...,ib, : :] is an M x N matrix.

LinearOperatorLowRankUpdate represents A = L + U D V^H, where

L, is a LinearOperator representing [batch] M x N matrices  
U, is a [batch] M x K matrix.  Typically K << M.  
D, is a [batch] K x K matrix.  
V, is a [batch] N x K matrix.  Typically K << N.  
V^H is the Hermitian transpose (adjoint) of V.

If M = N, determinants and solves are done using the matrix determinant lemma and Woodbury identities, and thus require L and D to be non-singular.

Solves and determinants will be attempted unless the "is\_non\_singular" property of L and D is False.

In the event that L and D are positive-definite, and U = V, solves and determinants can be done using a Cholesky factorization.

# Create a 3 x 3 diagonal linear operator.  
diag\_operator = LinearOperatorDiag(  
    diag\_update=[1., 2., 3.], is\_non\_singular=True, is\_self\_adjoint=True,  
    is\_positive\_definite=True)  
  
# Perturb with a rank 2 perturbation  
operator = LinearOperatorLowRankUpdate(  
    operator=diag\_operator,  
    u=[[1., 2.], [-1., 3.], [0., 0.]],  
    diag\_update=[11., 12.],  
    v=[[1., 2.], [-1., 3.], [10., 10.]])  
  
operator.shape  
==> [3, 3]  
  
operator.log\_abs\_determinant()  
==> scalar Tensor  
  
x = ... Shape [3, 4] Tensor  
operator.matmul(x)  
==> Shape [3, 4] Tensor

### Shape compatibility

This operator acts on [batch] matrix with compatible shape. x is a batch matrix with compatible shape for matmul and solve if

operator.shape = [B1,...,Bb] + [M, N],  with b >= 0  
x.shape =        [B1,...,Bb] + [N, R],  with R >= 0.

### Performance

Suppose operator is a LinearOperatorLowRankUpdate of shape [M, N], made from a rank Kupdate of base\_operator which performs .matmul(x) on x having x.shape = [N, R] with O(L\_matmul\*N\*R) complexity (and similarly for solve, determinant. Then, if x.shape = [N, R],

* operator.matmul(x) is O(L\_matmul\*N\*R + K\*N\*R)

and if M = N,

* operator.solve(x) is O(L\_matmul\*N\*R + N\*K\*R + K^2\*R + K^3)
* operator.determinant() is O(L\_determinant + L\_solve\*N\*K + K^2\*N + K^3)

If instead operator and x have shape [B1,...,Bb, M, N] and [B1,...,Bb, N, R], every operation increases in complexity by B1\*...\*Bb.

#### Matrix property hints

This LinearOperator is initialized with boolean flags of the form is\_X, for X = non\_singular, self\_adjoint, positive\_definite, diag\_update\_positive and square. These have the following meaning:

* If is\_X == True, callers should expect the operator to have the property X. This is a promise that should be fulfilled, but is not a runtime assert. For example, finite floating point precision may result in these promises being violated.
* If is\_X == False, callers should expect the operator to not have X.
* If is\_X == None (the default), callers should have no expectation either way.

## \_\_init\_\_

\_\_init\_\_(  
    base\_operator,  
    u,  
    diag\_update=None,  
    v=None,  
    is\_diag\_update\_positive=None,  
    is\_non\_singular=None,  
    is\_self\_adjoint=None,  
    is\_positive\_definite=None,  
    is\_square=None,  
    name='LinearOperatorLowRankUpdate'  
)

Initialize a LinearOperatorLowRankUpdate.

This creates a LinearOperator of the form A = L + U D V^H, with L a LinearOperator, U, Vboth [batch] matrices, and D a [batch] diagonal matrix.

If L is non-singular, solves and determinants are available. Solves/determinants both involve a solve/determinant of a K x K system. In the event that L and D are self-adjoint positive-definite, and U = V, this can be done using a Cholesky factorization. The user should set the is\_X matrix property hints, which will trigger the appropriate code path.

#### Args:

* **base\_operator**: Shape [B1,...,Bb, M, N].
* **u**: Shape [B1,...,Bb, M, K] Tensor of same dtype as base\_operator. This is U above.
* **diag\_update**: Optional shape [B1,...,Bb, K] Tensor with same dtype as base\_operator. This is the diagonal of D above. Defaults to D being the identity operator.
* **v**: Optional Tensor of same dtype as u and shape [B1,...,Bb, N, K] Defaults to v = u, in which case the perturbation is symmetric. If M != N, then v must be set since the perturbation is not square.
* **is\_diag\_update\_positive**: Python bool. If True, expect diag\_update > 0.
* **is\_non\_singular**: Expect that this operator is non-singular. Default is None, unless is\_positive\_definite is auto-set to be True (see below).
* **is\_self\_adjoint**: Expect that this operator is equal to its hermitian transpose. Default is None, unless base\_operator is self-adjoint and v = None (meaning u=v), in which case this defaults to True.
* **is\_positive\_definite**: Expect that this operator is positive definite. Default is None, unless base\_operator is positive-definite v = None (meaning u=v), and is\_diag\_update\_positive, in which case this defaults to True. Note that we say an operator is positive definite when the quadratic form x^H A x has positive real part for all nonzero x.
* **is\_square**: Expect that this operator acts like square [batch] matrices.
* **name**: A name for this LinearOperator.

#### Raises:

* **ValueError**: If is\_X flags are set in an inconsistent way.

## Properties

### H

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### base\_operator

If this operator is A = L + U D V^H, this is the L.

### batch\_shape

TensorShape of batch dimensions of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb]), equivalent to A.get\_shape()[:-2]

#### Returns:

TensorShape, statically determined, may be undefined.

### diag\_operator

If this operator is A = L + U D V^H, this is D.

### diag\_update

If this operator is A = L + U D V^H, this is the diagonal of D.

### domain\_dimension

Dimension (in the sense of vector spaces) of the domain of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Returns:

Dimension object.

### dtype

The DType of Tensors handled by this LinearOperator.

### graph\_parents

List of graph dependencies of this LinearOperator.

### is\_diag\_update\_positive

If this operator is A = L + U D V^H, this hints D > 0 elementwise.

### is\_non\_singular

### is\_positive\_definite

### is\_self\_adjoint

### is\_square

Return True/False depending on if this operator is square.

### name

Name prepended to all ops created by this LinearOperator.

### range\_dimension

Dimension (in the sense of vector spaces) of the range of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Returns:

Dimension object.

### shape

TensorShape of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb, M, N]), equivalent to A.get\_shape().

#### Returns:

TensorShape, statically determined, may be undefined.

### tensor\_rank

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

Python integer, or None if the tensor rank is undefined.

### u

If this operator is A = L + U D V^H, this is the U.

### v

If this operator is A = L + U D V^H, this is the V.

## Methods

### add\_to\_tensor

add\_to\_tensor(  
    x,  
    name='add\_to\_tensor'  
)

Add matrix represented by this operator to x. Equivalent to A + x.

#### Args:

* **x**: Tensor with same dtype and shape broadcastable to self.shape.
* **name**: A name to give this Op.

#### Returns:

A Tensor with broadcast shape and same dtype as self.

### adjoint

adjoint(name='adjoint')

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### assert\_non\_singular

assert\_non\_singular(name='assert\_non\_singular')

Returns an Op that asserts this operator is non singular.

This operator is considered non-singular if

ConditionNumber < max{100, range\_dimension, domain\_dimension} \* eps,  
eps := np.finfo(self.dtype.as\_numpy\_dtype).eps

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is singular.

### assert\_positive\_definite

assert\_positive\_definite(name='assert\_positive\_definite')

Returns an Op that asserts this operator is positive definite.

Here, positive definite means that the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive definite.

#### Args:

* **name**: A name to give this Op.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not positive definite.

### assert\_self\_adjoint

assert\_self\_adjoint(name='assert\_self\_adjoint')

Returns an Op that asserts this operator is self-adjoint.

Here we check that this operator is exactly equal to its hermitian transpose.

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not self-adjoint.

### batch\_shape\_tensor

batch\_shape\_tensor(name='batch\_shape\_tensor')

Shape of batch dimensions of this operator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb].

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### cholesky

cholesky(name='cholesky')

Returns a Cholesky factor as a LinearOperator.

Given A representing this LinearOperator, if A is positive definite self-adjoint, return L, where A = L L^T, i.e. the cholesky decomposition.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the lower triangular matrix in the Cholesky decomposition.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be positive definite and self adjoint.

### determinant

determinant(name='det')

Determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### diag\_part

diag\_part(name='diag\_part')

Efficiently get the [batch] diagonal part of this operator.

If this operator has shape [B1,...,Bb, M, N], this returns a Tensor diagonal, of shape [B1,...,Bb, min(M, N)], where diagonal[b1,...,bb, i] = self.to\_dense()[b1,...,bb, i, i].

my\_operator = LinearOperatorDiag([1., 2.])  
  
# Efficiently get the diagonal  
my\_operator.diag\_part()  
==> [1., 2.]  
  
# Equivalent, but inefficient method  
tf.linalg.diag\_part(my\_operator.to\_dense())  
==> [1., 2.]

#### Args:

* **name**: A name for this Op.

#### Returns:

* **diag\_part**: A Tensor of same dtype as self.

### domain\_dimension\_tensor

domain\_dimension\_tensor(name='domain\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the domain of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### inverse

inverse(name='inverse')

Returns the Inverse of this LinearOperator.

Given A representing this LinearOperator, return a LinearOperator representing A^-1.

#### Args:

* **name**: A name scope to use for ops added by this method.

#### Returns:

LinearOperator representing inverse of this matrix.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be non\_singular.

### log\_abs\_determinant

log\_abs\_determinant(name='log\_abs\_det')

Log absolute value of determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### matmul

matmul(  
    x,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='matmul'  
)

Transform [batch] matrix x with left multiplication: x --> Ax.

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
X = ... # shape [..., N, R], batch matrix, R > 0.  
  
Y = operator.matmul(X)  
Y.shape  
==> [..., M, R]  
  
Y[..., :, r] = sum\_j A[..., :, j] X[j, r]

#### Args:

* **x**: LinearOperator or Tensor with compatible shape and same dtype as self. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **adjoint\_arg**: Python bool. If True, compute A x^H where x^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name for this Op.

#### Returns:

A LinearOperator or Tensor with shape [..., M, R] and same dtype as self.

### matvec

matvec(  
    x,  
    adjoint=False,  
    name='matvec'  
)

Transform [batch] vector x with left multiplication: x --> Ax.

# Make an operator acting like batch matric A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
  
X = ... # shape [..., N], batch vector  
  
Y = operator.matvec(X)  
Y.shape  
==> [..., M]  
  
Y[..., :] = sum\_j A[..., :, j] X[..., j]

#### Args:

* **x**: Tensor with compatible shape and same dtype as self. x is treated as a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **name**: A name for this Op.

#### Returns:

A Tensor with shape [..., M] and same dtype as self.

### range\_dimension\_tensor

range\_dimension\_tensor(name='range\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the range of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### shape\_tensor

shape\_tensor(name='shape\_tensor')

Shape of this LinearOperator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb, M, N], equivalent to tf.shape(A).

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### solve

solve(  
    rhs,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='solve'  
)

Solve (exact or approx) R (batch) systems of equations: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve R > 0 linear systems for every member of the batch.  
RHS = ... # shape [..., M, R]  
  
X = operator.solve(RHS)  
# X[..., :, r] is the solution to the r'th linear system  
# sum\_j A[..., :, j] X[..., j, r] = RHS[..., :, r]  
  
operator.matmul(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator and compatible shape. rhs is treated like a [batch] matrix meaning for every set of leading dimensions, the last two dimensions defines a matrix. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **adjoint\_arg**: Python bool. If True, solve A X = rhs^H where rhs^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N, R] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### solvevec

solvevec(  
    rhs,  
    adjoint=False,  
    name='solve'  
)

Solve single equation with best effort: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve one linear system for every member of the batch.  
RHS = ... # shape [..., M]  
  
X = operator.solvevec(RHS)  
# X is the solution to the linear system  
# sum\_j A[..., :, j] X[..., j] = RHS[..., :]  
  
operator.matvec(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator. rhs is treated like a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility regarding batch dimensions.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### tensor\_rank\_tensor

tensor\_rank\_tensor(name='tensor\_rank\_tensor')

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor, determined at runtime.

### to\_dense

to\_dense(name='to\_dense')

Return a dense (batch) matrix representing this operator.

### trace

trace(name='trace')

Trace of the linear operator, equal to sum of self.diag\_part().

If the operator is square, this is also the sum of the eigenvalues.

#### Args:

* **name**: A name for this Op.

#### Returns:

Shape [B1,...,Bb] Tensor of same dtype as self.

# tf.linalg.LinearOperatorScaledIdentity

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorScaledIdentity#top_of_page)
* [Class LinearOperatorScaledIdentity](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorScaledIdentity#class_linearoperatorscaledidentity)
  + [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorScaledIdentity#aliases)
  + [Shape compatibility](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorScaledIdentity#shape_compatibility)
  + [Performance](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorScaledIdentity#performance)

## Class LinearOperatorScaledIdentity

LinearOperator acting like a scaled [batch] identity matrix A = c I.

### Aliases:

* Class tf.compat.v1.linalg.LinearOperatorScaledIdentity
* Class tf.compat.v2.linalg.LinearOperatorScaledIdentity
* Class tf.linalg.LinearOperatorScaledIdentity

Defined in [python/ops/linalg/linear\_operator\_identity.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg/linear_operator_identity.py).

This operator acts like a scaled [batch] identity matrix A with shape [B1,...,Bb, N, N] for some b >= 0. The first b indices index a batch member. For every batch index (i1,...,ib), A[i1,...,ib, : :] is a scaled version of the N x N identity matrix.

LinearOperatorIdentity is initialized with num\_rows, and a multiplier (a Tensor) of shape [B1,...,Bb]. N is set to num\_rows, and the multiplier determines the scale for each batch member.

# Create a 2 x 2 scaled identity matrix.  
operator = LinearOperatorIdentity(num\_rows=2, multiplier=3.)  
  
operator.to\_dense()  
==> [[3., 0.]  
     [0., 3.]]  
  
operator.shape  
==> [2, 2]  
  
operator.log\_abs\_determinant()  
==> 2 \* Log[3]  
  
x = ... Shape [2, 4] Tensor  
operator.matmul(x)  
==> 3 \* x  
  
y = tf.random.normal(shape=[3, 2, 4])  
# Note that y.shape is compatible with operator.shape because operator.shape  
# is broadcast to [3, 2, 2].  
x = operator.solve(y)  
==> 3 \* x  
  
# Create a 2-batch of 2x2 identity matrices  
operator = LinearOperatorIdentity(num\_rows=2, multiplier=5.)  
operator.to\_dense()  
==> [[[5., 0.]  
      [0., 5.]],  
     [[5., 0.]  
      [0., 5.]]]  
  
x = ... Shape [2, 2, 3]  
operator.matmul(x)  
==> 5 \* x  
  
# Here the operator and x have different batch\_shape, and are broadcast.  
x = ... Shape [1, 2, 3]  
operator.matmul(x)  
==> 5 \* x

### Shape compatibility

This operator acts on [batch] matrix with compatible shape. x is a batch matrix with compatible shape for matmul and solve if

operator.shape = [B1,...,Bb] + [N, N],  with b >= 0  
x.shape =   [C1,...,Cc] + [N, R],  
and [C1,...,Cc] broadcasts with [B1,...,Bb] to [D1,...,Dd]

### Performance

* operator.matmul(x) is O(D1\*...\*Dd\*N\*R)
* operator.solve(x) is O(D1\*...\*Dd\*N\*R)
* operator.determinant() is O(D1\*...\*Dd)

#### Matrix property hints

This LinearOperator is initialized with boolean flags of the form is\_X, for X = non\_singular, self\_adjoint, positive\_definite, square. These have the following meaning \* If is\_X == True, callers should expect the operator to have the property X. This is a promise that should be fulfilled, but is not a runtime assert. For example, finite floating point precision may result in these promises being violated. \* If is\_X == False, callers should expect the operator to not have X. \* If is\_X == None (the default), callers should have no expectation either way.

## \_\_init\_\_

\_\_init\_\_(  
    num\_rows,  
    multiplier,  
    is\_non\_singular=None,  
    is\_self\_adjoint=None,  
    is\_positive\_definite=None,  
    is\_square=True,  
    assert\_proper\_shapes=False,  
    name='LinearOperatorScaledIdentity'  
)

Initialize a LinearOperatorScaledIdentity.

The LinearOperatorScaledIdentity is initialized with num\_rows, which determines the size of each identity matrix, and a multiplier, which defines dtype, batch shape, and scale of each matrix.

This operator is able to broadcast the leading (batch) dimensions.

#### Args:

* **num\_rows**: Scalar non-negative integer Tensor. Number of rows in the corresponding identity matrix.
* **multiplier**: Tensor of shape [B1,...,Bb], or [] (a scalar).
* **is\_non\_singular**: Expect that this operator is non-singular.
* **is\_self\_adjoint**: Expect that this operator is equal to its hermitian transpose.
* **is\_positive\_definite**: Expect that this operator is positive definite, meaning the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive-definite. See: https://en.wikipedia.org/wiki/Positive-definite\_matrix#Extension\_for\_non-symmetric\_matrices
* **is\_square**: Expect that this operator acts like square [batch] matrices.
* **assert\_proper\_shapes**: Python bool. If False, only perform static checks that initialization and method arguments have proper shape. If True, and static checks are inconclusive, add asserts to the graph.
* **name**: A name for this LinearOperator

#### Raises:

* **ValueError**: If num\_rows is determined statically to be non-scalar, or negative.

## Properties

### H

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### batch\_shape

TensorShape of batch dimensions of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb]), equivalent to A.get\_shape()[:-2]

#### Returns:

TensorShape, statically determined, may be undefined.

### domain\_dimension

Dimension (in the sense of vector spaces) of the domain of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Returns:

Dimension object.

### dtype

The DType of Tensors handled by this LinearOperator.

### graph\_parents

List of graph dependencies of this LinearOperator.

### is\_non\_singular

### is\_positive\_definite

### is\_self\_adjoint

### is\_square

Return True/False depending on if this operator is square.

### multiplier

The [batch] scalar Tensor, c in cI.

### name

Name prepended to all ops created by this LinearOperator.

### range\_dimension

Dimension (in the sense of vector spaces) of the range of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Returns:

Dimension object.

### shape

TensorShape of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb, M, N]), equivalent to A.get\_shape().

#### Returns:

TensorShape, statically determined, may be undefined.

### tensor\_rank

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

Python integer, or None if the tensor rank is undefined.

## Methods

### add\_to\_tensor

add\_to\_tensor(  
    mat,  
    name='add\_to\_tensor'  
)

Add matrix represented by this operator to mat. Equiv to I + mat.

#### Args:

* **mat**: Tensor with same dtype and shape broadcastable to self.
* **name**: A name to give this Op.

#### Returns:

A Tensor with broadcast shape and same dtype as self.

### adjoint

adjoint(name='adjoint')

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### assert\_non\_singular

assert\_non\_singular(name='assert\_non\_singular')

Returns an Op that asserts this operator is non singular.

This operator is considered non-singular if

ConditionNumber < max{100, range\_dimension, domain\_dimension} \* eps,  
eps := np.finfo(self.dtype.as\_numpy\_dtype).eps

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is singular.

### assert\_positive\_definite

assert\_positive\_definite(name='assert\_positive\_definite')

Returns an Op that asserts this operator is positive definite.

Here, positive definite means that the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive definite.

#### Args:

* **name**: A name to give this Op.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not positive definite.

### assert\_self\_adjoint

assert\_self\_adjoint(name='assert\_self\_adjoint')

Returns an Op that asserts this operator is self-adjoint.

Here we check that this operator is exactly equal to its hermitian transpose.

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not self-adjoint.

### batch\_shape\_tensor

batch\_shape\_tensor(name='batch\_shape\_tensor')

Shape of batch dimensions of this operator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb].

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### cholesky

cholesky(name='cholesky')

Returns a Cholesky factor as a LinearOperator.

Given A representing this LinearOperator, if A is positive definite self-adjoint, return L, where A = L L^T, i.e. the cholesky decomposition.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the lower triangular matrix in the Cholesky decomposition.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be positive definite and self adjoint.

### determinant

determinant(name='det')

Determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### diag\_part

diag\_part(name='diag\_part')

Efficiently get the [batch] diagonal part of this operator.

If this operator has shape [B1,...,Bb, M, N], this returns a Tensor diagonal, of shape [B1,...,Bb, min(M, N)], where diagonal[b1,...,bb, i] = self.to\_dense()[b1,...,bb, i, i].

my\_operator = LinearOperatorDiag([1., 2.])  
  
# Efficiently get the diagonal  
my\_operator.diag\_part()  
==> [1., 2.]  
  
# Equivalent, but inefficient method  
tf.linalg.diag\_part(my\_operator.to\_dense())  
==> [1., 2.]

#### Args:

* **name**: A name for this Op.

#### Returns:

* **diag\_part**: A Tensor of same dtype as self.

### domain\_dimension\_tensor

domain\_dimension\_tensor(name='domain\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the domain of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### inverse

inverse(name='inverse')

Returns the Inverse of this LinearOperator.

Given A representing this LinearOperator, return a LinearOperator representing A^-1.

#### Args:

* **name**: A name scope to use for ops added by this method.

#### Returns:

LinearOperator representing inverse of this matrix.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be non\_singular.

### log\_abs\_determinant

log\_abs\_determinant(name='log\_abs\_det')

Log absolute value of determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### matmul

matmul(  
    x,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='matmul'  
)

Transform [batch] matrix x with left multiplication: x --> Ax.

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
X = ... # shape [..., N, R], batch matrix, R > 0.  
  
Y = operator.matmul(X)  
Y.shape  
==> [..., M, R]  
  
Y[..., :, r] = sum\_j A[..., :, j] X[j, r]

#### Args:

* **x**: LinearOperator or Tensor with compatible shape and same dtype as self. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **adjoint\_arg**: Python bool. If True, compute A x^H where x^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name for this Op.

#### Returns:

A LinearOperator or Tensor with shape [..., M, R] and same dtype as self.

### matvec

matvec(  
    x,  
    adjoint=False,  
    name='matvec'  
)

Transform [batch] vector x with left multiplication: x --> Ax.

# Make an operator acting like batch matric A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
  
X = ... # shape [..., N], batch vector  
  
Y = operator.matvec(X)  
Y.shape  
==> [..., M]  
  
Y[..., :] = sum\_j A[..., :, j] X[..., j]

#### Args:

* **x**: Tensor with compatible shape and same dtype as self. x is treated as a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **name**: A name for this Op.

#### Returns:

A Tensor with shape [..., M] and same dtype as self.

### range\_dimension\_tensor

range\_dimension\_tensor(name='range\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the range of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### shape\_tensor

shape\_tensor(name='shape\_tensor')

Shape of this LinearOperator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb, M, N], equivalent to tf.shape(A).

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### solve

solve(  
    rhs,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='solve'  
)

Solve (exact or approx) R (batch) systems of equations: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve R > 0 linear systems for every member of the batch.  
RHS = ... # shape [..., M, R]  
  
X = operator.solve(RHS)  
# X[..., :, r] is the solution to the r'th linear system  
# sum\_j A[..., :, j] X[..., j, r] = RHS[..., :, r]  
  
operator.matmul(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator and compatible shape. rhs is treated like a [batch] matrix meaning for every set of leading dimensions, the last two dimensions defines a matrix. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **adjoint\_arg**: Python bool. If True, solve A X = rhs^H where rhs^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N, R] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### solvevec

solvevec(  
    rhs,  
    adjoint=False,  
    name='solve'  
)

Solve single equation with best effort: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve one linear system for every member of the batch.  
RHS = ... # shape [..., M]  
  
X = operator.solvevec(RHS)  
# X is the solution to the linear system  
# sum\_j A[..., :, j] X[..., j] = RHS[..., :]  
  
operator.matvec(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator. rhs is treated like a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility regarding batch dimensions.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### tensor\_rank\_tensor

tensor\_rank\_tensor(name='tensor\_rank\_tensor')

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor, determined at runtime.

### to\_dense

to\_dense(name='to\_dense')

Return a dense (batch) matrix representing this operator.

### trace

trace(name='trace')

Trace of the linear operator, equal to sum of self.diag\_part().

If the operator is square, this is also the sum of the eigenvalues.

#### Args:

* **name**: A name for this Op.

#### Returns:

Shape [B1,...,Bb] Tensor of same dtype as self.

# tf.linalg.LinearOperatorToeplitz

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorToeplitz#top_of_page)
* [Class LinearOperatorToeplitz](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorToeplitz#class_linearoperatortoeplitz)
  + [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorToeplitz#aliases)
* [Properties](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorToeplitz#properties)
  + [H](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorToeplitz#H)

## Class LinearOperatorToeplitz

LinearOperator acting like a [batch] of toeplitz matrices.

Inherits From: [LinearOperator](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperator)

### Aliases:

* Class tf.compat.v1.linalg.LinearOperatorToeplitz
* Class tf.compat.v2.linalg.LinearOperatorToeplitz
* Class tf.linalg.LinearOperatorToeplitz

Defined in [python/ops/linalg/linear\_operator\_toeplitz.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg/linear_operator_toeplitz.py).

This operator acts like a [batch] Toeplitz matrix A with shape [B1,...,Bb, N, N] for some b >= 0. The first b indices index a batch member. For every batch index (i1,...,ib), A[i1,...,ib, : :] is an N x N matrix. This matrix A is not materialized, but for purposes of broadcasting this shape will be relevant.

#### Description in terms of toeplitz matrices

Toeplitz means that A has constant diagonals. Hence, A can be generated with two vectors. One represents the first column of the matrix, and the other represents the first row.

Below is a 4 x 4 example:

A = |a b c d|  
    |e a b c|  
    |f e a b|  
    |g f e a|

#### Example of a Toeplitz operator.

# Create a 3 x 3 Toeplitz operator.  
col = [1., 2., 3.]  
row = [1., 4., -9.]  
operator = LinearOperatorToeplitz(col, row)  
  
operator.to\_dense()  
==> [[1., 4., -9.],  
     [2., 1., 4.],  
     [3., 2., 1.]]  
  
operator.shape  
==> [3, 3]  
  
operator.log\_abs\_determinant()  
==> scalar Tensor  
  
x = ... Shape [3, 4] Tensor  
operator.matmul(x)  
==> Shape [3, 4] Tensor  
  
#### Shape compatibility  
  
This operator acts on [batch] matrix with compatible shape.  
`x` is a batch matrix with compatible shape for `matmul` and `solve` if

operator.shape = [B1,...,Bb] + [N, N], with b >= 0 x.shape = [C1,...,Cc] + [N, R], and [C1,...,Cc] broadcasts with [B1,...,Bb] to [D1,...,Dd]

#### Matrix property hints  
  
This `LinearOperator` is initialized with boolean flags of the form `is\_X`,  
for `X = non\_singular, self\_adjoint, positive\_definite, square`.  
These have the following meaning:  
  
\* If `is\_X == True`, callers should expect the operator to have the  
  property `X`.  This is a promise that should be fulfilled, but is \*not\* a  
  runtime assert.  For example, finite floating point precision may result  
  in these promises being violated.  
\* If `is\_X == False`, callers should expect the operator to not have `X`.  
\* If `is\_X == None` (the default), callers should have no expectation either  
  way.  
  
<h2 id="\_\_init\_\_"><code>\_\_init\_\_</code></h2>  
  
``` python  
\_\_init\_\_(  
    col,  
    row,  
    is\_non\_singular=None,  
    is\_self\_adjoint=None,  
    is\_positive\_definite=None,  
    is\_square=None,  
    name='LinearOperatorToeplitz'  
)

Initialize a LinearOperatorToeplitz.

#### Args:

* **col**: Shape [B1,...,Bb, N] Tensor with b >= 0 N >= 0. The first column of the operator. Allowed dtypes: float16, float32, float64, complex64, complex128. Note that the first entry of col is assumed to be the same as the first entry of row.
* **row**: Shape [B1,...,Bb, N] Tensor with b >= 0 N >= 0. The first row of the operator. Allowed dtypes: float16, float32, float64, complex64, complex128. Note that the first entry of row is assumed to be the same as the first entry of col.
* **is\_non\_singular**: Expect that this operator is non-singular.
* **is\_self\_adjoint**: Expect that this operator is equal to its hermitian transpose. If diag.dtypeis real, this is auto-set to True.
* **is\_positive\_definite**: Expect that this operator is positive definite, meaning the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive-definite. See: https://en.wikipedia.org/wiki/Positive-definite\_matrix#Extension\_for\_non-symmetric\_matrices
* **is\_square**: Expect that this operator acts like square [batch] matrices.
* **name**: A name for this LinearOperator.

## Properties

### H

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### batch\_shape

TensorShape of batch dimensions of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb]), equivalent to A.get\_shape()[:-2]

#### Returns:

TensorShape, statically determined, may be undefined.

### col

### domain\_dimension

Dimension (in the sense of vector spaces) of the domain of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Returns:

Dimension object.

### dtype

The DType of Tensors handled by this LinearOperator.

### graph\_parents

List of graph dependencies of this LinearOperator.

### is\_non\_singular

### is\_positive\_definite

### is\_self\_adjoint

### is\_square

Return True/False depending on if this operator is square.

### name

Name prepended to all ops created by this LinearOperator.

### range\_dimension

Dimension (in the sense of vector spaces) of the range of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Returns:

Dimension object.

### row

### shape

TensorShape of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb, M, N]), equivalent to A.get\_shape().

#### Returns:

TensorShape, statically determined, may be undefined.

### tensor\_rank

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

Python integer, or None if the tensor rank is undefined.

## Methods

### add\_to\_tensor

add\_to\_tensor(  
    x,  
    name='add\_to\_tensor'  
)

Add matrix represented by this operator to x. Equivalent to A + x.

#### Args:

* **x**: Tensor with same dtype and shape broadcastable to self.shape.
* **name**: A name to give this Op.

#### Returns:

A Tensor with broadcast shape and same dtype as self.

### adjoint

adjoint(name='adjoint')

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### assert\_non\_singular

assert\_non\_singular(name='assert\_non\_singular')

Returns an Op that asserts this operator is non singular.

This operator is considered non-singular if

ConditionNumber < max{100, range\_dimension, domain\_dimension} \* eps,  
eps := np.finfo(self.dtype.as\_numpy\_dtype).eps

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is singular.

### assert\_positive\_definite

assert\_positive\_definite(name='assert\_positive\_definite')

Returns an Op that asserts this operator is positive definite.

Here, positive definite means that the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive definite.

#### Args:

* **name**: A name to give this Op.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not positive definite.

### assert\_self\_adjoint

assert\_self\_adjoint(name='assert\_self\_adjoint')

Returns an Op that asserts this operator is self-adjoint.

Here we check that this operator is exactly equal to its hermitian transpose.

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not self-adjoint.

### batch\_shape\_tensor

batch\_shape\_tensor(name='batch\_shape\_tensor')

Shape of batch dimensions of this operator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb].

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### cholesky

cholesky(name='cholesky')

Returns a Cholesky factor as a LinearOperator.

Given A representing this LinearOperator, if A is positive definite self-adjoint, return L, where A = L L^T, i.e. the cholesky decomposition.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the lower triangular matrix in the Cholesky decomposition.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be positive definite and self adjoint.

### determinant

determinant(name='det')

Determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### diag\_part

diag\_part(name='diag\_part')

Efficiently get the [batch] diagonal part of this operator.

If this operator has shape [B1,...,Bb, M, N], this returns a Tensor diagonal, of shape [B1,...,Bb, min(M, N)], where diagonal[b1,...,bb, i] = self.to\_dense()[b1,...,bb, i, i].

my\_operator = LinearOperatorDiag([1., 2.])  
  
# Efficiently get the diagonal  
my\_operator.diag\_part()  
==> [1., 2.]  
  
# Equivalent, but inefficient method  
tf.linalg.diag\_part(my\_operator.to\_dense())  
==> [1., 2.]

#### Args:

* **name**: A name for this Op.

#### Returns:

* **diag\_part**: A Tensor of same dtype as self.

### domain\_dimension\_tensor

domain\_dimension\_tensor(name='domain\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the domain of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### inverse

inverse(name='inverse')

Returns the Inverse of this LinearOperator.

Given A representing this LinearOperator, return a LinearOperator representing A^-1.

#### Args:

* **name**: A name scope to use for ops added by this method.

#### Returns:

LinearOperator representing inverse of this matrix.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be non\_singular.

### log\_abs\_determinant

log\_abs\_determinant(name='log\_abs\_det')

Log absolute value of determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### matmul

matmul(  
    x,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='matmul'  
)

Transform [batch] matrix x with left multiplication: x --> Ax.

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
X = ... # shape [..., N, R], batch matrix, R > 0.  
  
Y = operator.matmul(X)  
Y.shape  
==> [..., M, R]  
  
Y[..., :, r] = sum\_j A[..., :, j] X[j, r]

#### Args:

* **x**: LinearOperator or Tensor with compatible shape and same dtype as self. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **adjoint\_arg**: Python bool. If True, compute A x^H where x^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name for this Op.

#### Returns:

A LinearOperator or Tensor with shape [..., M, R] and same dtype as self.

### matvec

matvec(  
    x,  
    adjoint=False,  
    name='matvec'  
)

Transform [batch] vector x with left multiplication: x --> Ax.

# Make an operator acting like batch matric A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
  
X = ... # shape [..., N], batch vector  
  
Y = operator.matvec(X)  
Y.shape  
==> [..., M]  
  
Y[..., :] = sum\_j A[..., :, j] X[..., j]

#### Args:

* **x**: Tensor with compatible shape and same dtype as self. x is treated as a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **name**: A name for this Op.

#### Returns:

A Tensor with shape [..., M] and same dtype as self.

### range\_dimension\_tensor

range\_dimension\_tensor(name='range\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the range of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### shape\_tensor

shape\_tensor(name='shape\_tensor')

Shape of this LinearOperator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb, M, N], equivalent to tf.shape(A).

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### solve

solve(  
    rhs,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='solve'  
)

Solve (exact or approx) R (batch) systems of equations: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve R > 0 linear systems for every member of the batch.  
RHS = ... # shape [..., M, R]  
  
X = operator.solve(RHS)  
# X[..., :, r] is the solution to the r'th linear system  
# sum\_j A[..., :, j] X[..., j, r] = RHS[..., :, r]  
  
operator.matmul(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator and compatible shape. rhs is treated like a [batch] matrix meaning for every set of leading dimensions, the last two dimensions defines a matrix. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **adjoint\_arg**: Python bool. If True, solve A X = rhs^H where rhs^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N, R] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### solvevec

solvevec(  
    rhs,  
    adjoint=False,  
    name='solve'  
)

Solve single equation with best effort: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve one linear system for every member of the batch.  
RHS = ... # shape [..., M]  
  
X = operator.solvevec(RHS)  
# X is the solution to the linear system  
# sum\_j A[..., :, j] X[..., j] = RHS[..., :]  
  
operator.matvec(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator. rhs is treated like a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility regarding batch dimensions.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### tensor\_rank\_tensor

tensor\_rank\_tensor(name='tensor\_rank\_tensor')

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor, determined at runtime.

### to\_dense

to\_dense(name='to\_dense')

Return a dense (batch) matrix representing this operator.

### trace

trace(name='trace')

Trace of the linear operator, equal to sum of self.diag\_part().

If the operator is square, this is also the sum of the eigenvalues.

#### Args:

* **name**: A name for this Op.

#### Returns:

Shape [B1,...,Bb] Tensor of same dtype as self.

# tf.linalg.LinearOperatorZeros

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorZeros#top_of_page)
* [Class LinearOperatorZeros](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorZeros#class_linearoperatorzeros)
  + [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorZeros#aliases)
  + [Shape compatibility](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorZeros#shape_compatibility)
* [\_\_init\_\_](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperatorZeros#__init__)

## Class LinearOperatorZeros

LinearOperator acting like a [batch] zero matrix.

Inherits From: [LinearOperator](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/LinearOperator)

### Aliases:

* Class tf.compat.v1.linalg.LinearOperatorZeros
* Class tf.compat.v2.linalg.LinearOperatorZeros
* Class tf.linalg.LinearOperatorZeros

Defined in [python/ops/linalg/linear\_operator\_zeros.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg/linear_operator_zeros.py).

This operator acts like a [batch] zero matrix A with shape [B1,...,Bb, N, M] for some b >= 0. The first b indices index a batch member. For every batch index (i1,...,ib), A[i1,...,ib, : :]is an N x M matrix. This matrix A is not materialized, but for purposes of broadcasting this shape will be relevant.

LinearOperatorZeros is initialized with num\_rows, and optionally num\_columns,batch\_shape, anddtypearguments. Ifnum\_columnsisNone, then this operator will be initialized as a square matrix. Ifbatch\_shapeisNone, this operator efficiently passes through all arguments. Ifbatch\_shape` is provided, broadcasting may occur, which will require making copies.

# Create a 2 x 2 zero matrix.  
operator = LinearOperatorZero(num\_rows=2, dtype=tf.float32)  
  
operator.to\_dense()  
==> [[0., 0.]  
     [0., 0.]]  
  
operator.shape  
==> [2, 2]  
  
operator.determinant()  
==> 0.  
  
x = ... Shape [2, 4] Tensor  
operator.matmul(x)  
==> Shape [2, 4] Tensor, same as x.  
  
# Create a 2-batch of 2x2 zero matrices  
operator = LinearOperatorZeros(num\_rows=2, batch\_shape=[2])  
operator.to\_dense()  
==> [[[0., 0.]  
      [0., 0.]],  
     [[0., 0.]  
      [0., 0.]]]  
  
# Here, even though the operator has a batch shape, the input is the same as  
# the output, so x can be passed through without a copy.  The operator is able  
# to detect that no broadcast is necessary because both x and the operator  
# have statically defined shape.  
x = ... Shape [2, 2, 3]  
operator.matmul(x)  
==> Shape [2, 2, 3] Tensor, same as tf.zeros\_like(x)  
  
# Here the operator and x have different batch\_shape, and are broadcast.  
# This requires a copy, since the output is different size than the input.  
x = ... Shape [1, 2, 3]  
operator.matmul(x)  
==> Shape [2, 2, 3] Tensor, equal to tf.zeros\_like([x, x])

### Shape compatibility

This operator acts on [batch] matrix with compatible shape. x is a batch matrix with compatible shape for matmul and solve if

operator.shape = [B1,...,Bb] + [N, M],  with b >= 0  
x.shape =   [C1,...,Cc] + [M, R],  
and [C1,...,Cc] broadcasts with [B1,...,Bb] to [D1,...,Dd]

#### Matrix property hints

This LinearOperator is initialized with boolean flags of the form is\_X, for X = non\_singular, self\_adjoint, positive\_definite, square. These have the following meaning:

* If is\_X == True, callers should expect the operator to have the property X. This is a promise that should be fulfilled, but is not a runtime assert. For example, finite floating point precision may result in these promises being violated.
* If is\_X == False, callers should expect the operator to not have X.
* If is\_X == None (the default), callers should have no expectation either way.

## \_\_init\_\_

\_\_init\_\_(  
    num\_rows,  
    num\_columns=None,  
    batch\_shape=None,  
    dtype=None,  
    is\_non\_singular=False,  
    is\_self\_adjoint=True,  
    is\_positive\_definite=False,  
    is\_square=True,  
    assert\_proper\_shapes=False,  
    name='LinearOperatorZeros'  
)

Initialize a LinearOperatorZeros.

The LinearOperatorZeros is initialized with arguments defining dtype and shape.

This operator is able to broadcast the leading (batch) dimensions, which sometimes requires copying data. If batch\_shape is None, the operator can take arguments of any batch shape without copying. See examples.

#### Args:

* **num\_rows**: Scalar non-negative integer Tensor. Number of rows in the corresponding zero matrix.
* **num\_columns**: Scalar non-negative integer Tensor. Number of columns in the corresponding zero matrix. If None, defaults to the value of num\_rows.
* **batch\_shape**: Optional 1-D integer Tensor. The shape of the leading dimensions. If None, this operator has no leading dimensions.
* **dtype**: Data type of the matrix that this operator represents.
* **is\_non\_singular**: Expect that this operator is non-singular.
* **is\_self\_adjoint**: Expect that this operator is equal to its hermitian transpose.
* **is\_positive\_definite**: Expect that this operator is positive definite, meaning the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive-definite. See: https://en.wikipedia.org/wiki/Positive-definite\_matrix#Extension\_for\_non-symmetric\_matrices
* **is\_square**: Expect that this operator acts like square [batch] matrices.
* **assert\_proper\_shapes**: Python bool. If False, only perform static checks that initialization and method arguments have proper shape. If True, and static checks are inconclusive, add asserts to the graph.
* **name**: A name for this LinearOperator

#### Raises:

* **ValueError**: If num\_rows is determined statically to be non-scalar, or negative.
* **ValueError**: If num\_columns is determined statically to be non-scalar, or negative.
* **ValueError**: If batch\_shape is determined statically to not be 1-D, or negative.
* **ValueError**: If any of the following is not True: {is\_self\_adjoint, is\_non\_singular, is\_positive\_definite}.

## Properties

### H

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### batch\_shape

TensorShape of batch dimensions of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb]), equivalent to A.get\_shape()[:-2]

#### Returns:

TensorShape, statically determined, may be undefined.

### domain\_dimension

Dimension (in the sense of vector spaces) of the domain of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Returns:

Dimension object.

### dtype

The DType of Tensors handled by this LinearOperator.

### graph\_parents

List of graph dependencies of this LinearOperator.

### is\_non\_singular

### is\_positive\_definite

### is\_self\_adjoint

### is\_square

Return True/False depending on if this operator is square.

### name

Name prepended to all ops created by this LinearOperator.

### range\_dimension

Dimension (in the sense of vector spaces) of the range of this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Returns:

Dimension object.

### shape

TensorShape of this LinearOperator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returnsTensorShape([B1,...,Bb, M, N]), equivalent to A.get\_shape().

#### Returns:

TensorShape, statically determined, may be undefined.

### tensor\_rank

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

Python integer, or None if the tensor rank is undefined.

## Methods

### add\_to\_tensor

add\_to\_tensor(  
    mat,  
    name='add\_to\_tensor'  
)

Add matrix represented by this operator to mat. Equiv to I + mat.

#### Args:

* **mat**: Tensor with same dtype and shape broadcastable to self.
* **name**: A name to give this Op.

#### Returns:

A Tensor with broadcast shape and same dtype as self.

### adjoint

adjoint(name='adjoint')

Returns the adjoint of the current LinearOperator.

Given A representing this LinearOperator, return A\*. Note that calling self.adjoint() and self.H are equivalent.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the adjoint of this LinearOperator.

### assert\_non\_singular

assert\_non\_singular(name='assert\_non\_singular')

Returns an Op that asserts this operator is non singular.

This operator is considered non-singular if

ConditionNumber < max{100, range\_dimension, domain\_dimension} \* eps,  
eps := np.finfo(self.dtype.as\_numpy\_dtype).eps

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is singular.

### assert\_positive\_definite

assert\_positive\_definite(name='assert\_positive\_definite')

Returns an Op that asserts this operator is positive definite.

Here, positive definite means that the quadratic form x^H A x has positive real part for all nonzero x. Note that we do not require the operator to be self-adjoint to be positive definite.

#### Args:

* **name**: A name to give this Op.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not positive definite.

### assert\_self\_adjoint

assert\_self\_adjoint(name='assert\_self\_adjoint')

Returns an Op that asserts this operator is self-adjoint.

Here we check that this operator is exactly equal to its hermitian transpose.

#### Args:

* **name**: A string name to prepend to created ops.

#### Returns:

An Assert Op, that, when run, will raise an InvalidArgumentError if the operator is not self-adjoint.

### batch\_shape\_tensor

batch\_shape\_tensor(name='batch\_shape\_tensor')

Shape of batch dimensions of this operator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb].

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### cholesky

cholesky(name='cholesky')

Returns a Cholesky factor as a LinearOperator.

Given A representing this LinearOperator, if A is positive definite self-adjoint, return L, where A = L L^T, i.e. the cholesky decomposition.

#### Args:

* **name**: A name for this Op.

#### Returns:

LinearOperator which represents the lower triangular matrix in the Cholesky decomposition.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be positive definite and self adjoint.

### determinant

determinant(name='det')

Determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### diag\_part

diag\_part(name='diag\_part')

Efficiently get the [batch] diagonal part of this operator.

If this operator has shape [B1,...,Bb, M, N], this returns a Tensor diagonal, of shape [B1,...,Bb, min(M, N)], where diagonal[b1,...,bb, i] = self.to\_dense()[b1,...,bb, i, i].

my\_operator = LinearOperatorDiag([1., 2.])  
  
# Efficiently get the diagonal  
my\_operator.diag\_part()  
==> [1., 2.]  
  
# Equivalent, but inefficient method  
tf.linalg.diag\_part(my\_operator.to\_dense())  
==> [1., 2.]

#### Args:

* **name**: A name for this Op.

#### Returns:

* **diag\_part**: A Tensor of same dtype as self.

### domain\_dimension\_tensor

domain\_dimension\_tensor(name='domain\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the domain of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns N.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### inverse

inverse(name='inverse')

Returns the Inverse of this LinearOperator.

Given A representing this LinearOperator, return a LinearOperator representing A^-1.

#### Args:

* **name**: A name scope to use for ops added by this method.

#### Returns:

LinearOperator representing inverse of this matrix.

#### Raises:

* **ValueError**: When the LinearOperator is not hinted to be non\_singular.

### log\_abs\_determinant

log\_abs\_determinant(name='log\_abs\_det')

Log absolute value of determinant for every batch member.

#### Args:

* **name**: A name for this Op.

#### Returns:

Tensor with shape self.batch\_shape and same dtype as self.

#### Raises:

* **NotImplementedError**: If self.is\_square is False.

### matmul

matmul(  
    x,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='matmul'  
)

Transform [batch] matrix x with left multiplication: x --> Ax.

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
X = ... # shape [..., N, R], batch matrix, R > 0.  
  
Y = operator.matmul(X)  
Y.shape  
==> [..., M, R]  
  
Y[..., :, r] = sum\_j A[..., :, j] X[j, r]

#### Args:

* **x**: LinearOperator or Tensor with compatible shape and same dtype as self. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **adjoint\_arg**: Python bool. If True, compute A x^H where x^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name for this Op.

#### Returns:

A LinearOperator or Tensor with shape [..., M, R] and same dtype as self.

### matvec

matvec(  
    x,  
    adjoint=False,  
    name='matvec'  
)

Transform [batch] vector x with left multiplication: x --> Ax.

# Make an operator acting like batch matric A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
  
X = ... # shape [..., N], batch vector  
  
Y = operator.matvec(X)  
Y.shape  
==> [..., M]  
  
Y[..., :] = sum\_j A[..., :, j] X[..., j]

#### Args:

* **x**: Tensor with compatible shape and same dtype as self. x is treated as a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, left multiply by the adjoint: A^H x.
* **name**: A name for this Op.

#### Returns:

A Tensor with shape [..., M] and same dtype as self.

### range\_dimension\_tensor

range\_dimension\_tensor(name='range\_dimension\_tensor')

Dimension (in the sense of vector spaces) of the range of this operator.

Determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns M.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### shape\_tensor

shape\_tensor(name='shape\_tensor')

Shape of this LinearOperator, determined at runtime.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns a Tensor holding [B1,...,Bb, M, N], equivalent to tf.shape(A).

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor

### solve

solve(  
    rhs,  
    adjoint=False,  
    adjoint\_arg=False,  
    name='solve'  
)

Solve (exact or approx) R (batch) systems of equations: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve R > 0 linear systems for every member of the batch.  
RHS = ... # shape [..., M, R]  
  
X = operator.solve(RHS)  
# X[..., :, r] is the solution to the r'th linear system  
# sum\_j A[..., :, j] X[..., j, r] = RHS[..., :, r]  
  
operator.matmul(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator and compatible shape. rhs is treated like a [batch] matrix meaning for every set of leading dimensions, the last two dimensions defines a matrix. See class docstring for definition of compatibility.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **adjoint\_arg**: Python bool. If True, solve A X = rhs^H where rhs^H is the hermitian transpose (transposition and complex conjugation).
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N, R] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### solvevec

solvevec(  
    rhs,  
    adjoint=False,  
    name='solve'  
)

Solve single equation with best effort: A X = rhs.

The returned Tensor will be close to an exact solution if A is well conditioned. Otherwise closeness will vary. See class docstring for details.

#### Examples:

# Make an operator acting like batch matrix A.  Assume A.shape = [..., M, N]  
operator = LinearOperator(...)  
operator.shape = [..., M, N]  
  
# Solve one linear system for every member of the batch.  
RHS = ... # shape [..., M]  
  
X = operator.solvevec(RHS)  
# X is the solution to the linear system  
# sum\_j A[..., :, j] X[..., j] = RHS[..., :]  
  
operator.matvec(X)  
==> RHS

#### Args:

* **rhs**: Tensor with same dtype as this operator. rhs is treated like a [batch] vector meaning for every set of leading dimensions, the last dimension defines a vector. See class docstring for definition of compatibility regarding batch dimensions.
* **adjoint**: Python bool. If True, solve the system involving the adjoint of this LinearOperator: A^H X = rhs.
* **name**: A name scope to use for ops added by this method.

#### Returns:

Tensor with shape [...,N] and same dtype as rhs.

#### Raises:

* **NotImplementedError**: If self.is\_non\_singular or is\_square is False.

### tensor\_rank\_tensor

tensor\_rank\_tensor(name='tensor\_rank\_tensor')

Rank (in the sense of tensors) of matrix corresponding to this operator.

If this operator acts like the batch matrix A with A.shape = [B1,...,Bb, M, N], then this returns b + 2.

#### Args:

* **name**: A name for this Op.

#### Returns:

int32 Tensor, determined at runtime.

### to\_dense

to\_dense(name='to\_dense')

Return a dense (batch) matrix representing this operator.

### trace

trace(name='trace')

Trace of the linear operator, equal to sum of self.diag\_part().

If the operator is square, this is also the sum of the eigenvalues.

#### Args:

* **name**: A name for this Op.

#### Returns:

Shape [B1,...,Bb] Tensor of same dtype as self.

# tf.linalg.logdet

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/logdet#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/logdet#aliases)

Computes log of the determinant of a hermitian positive definite matrix.

### Aliases:

* tf.compat.v1.linalg.logdet
* tf.compat.v2.linalg.logdet
* tf.linalg.logdet

tf.linalg.logdet(  
    matrix,  
    name=None  
)

Defined in [python/ops/linalg/linalg\_impl.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg/linalg_impl.py).

# Compute the determinant of a matrix while reducing the chance of over- or  
underflow:  
A = ... # shape 10 x 10  
det = tf.exp(tf.linalg.logdet(A))  # scalar

#### Args:

* **matrix**: A Tensor. Must be float16, float32, float64, complex64, or complex128 with shape [..., M, M].
* **name**: A name to give this Op. Defaults to logdet.

#### Returns:

The natural log of the determinant of matrix.

#### Numpy Compatibility

Equivalent to numpy.linalg.slogdet, although no sign is returned since only hermitian positive definite matrices are supported.

# tf.linalg.logm

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/logm#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/logm#aliases)

Computes the matrix logarithm of one or more square matrices:

### Aliases:

* tf.compat.v1.linalg.logm
* tf.compat.v2.linalg.logm
* tf.linalg.logm

tf.linalg.logm(  
    input,  
    name=None  
)

Defined in generated file: python/ops/gen\_linalg\_ops.py.

log(exp(A))=A

This op is only defined for complex matrices. If A is positive-definite and real, then casting to a complex matrix, taking the logarithm and casting back to a real matrix will give the correct result.

This function computes the matrix logarithm using the Schur-Parlett algorithm. Details of the algorithm can be found in Section 11.6.2 of: Nicholas J. Higham, Functions of Matrices: Theory and Computation, SIAM 2008. ISBN 978-0-898716-46-7.

The input is a tensor of shape [..., M, M] whose inner-most 2 dimensions form square matrices. The output is a tensor of the same shape as the input containing the exponential for all input submatrices [..., :, :].

#### Args:

* **input**: A Tensor. Must be one of the following types: complex64, complex128. Shape is [..., M, M].
* **name**: A name for the operation (optional).

#### Returns:

A Tensor. Has the same type as input.

# tf.linalg.lstsq

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/lstsq#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/lstsq#aliases)

Solves one or more linear least-squares problems.

### Aliases:

* tf.compat.v1.linalg.lstsq
* tf.compat.v1.matrix\_solve\_ls
* tf.compat.v2.linalg.lstsq
* tf.linalg.lstsq

tf.linalg.lstsq(  
    matrix,  
    rhs,  
    l2\_regularizer=0.0,  
    fast=True,  
    name=None  
)

Defined in [python/ops/linalg\_ops.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg_ops.py).

matrix is a tensor of shape [..., M, N] whose inner-most 2 dimensions form M-by-N matrices. Rhs is a tensor of shape [..., M, K] whose inner-most 2 dimensions form M-by-K matrices. The computed output is a Tensor of shape [..., N, K] whose inner-most 2 dimensions form M-by-Kmatrices that solve the equations matrix[..., :, :] \* output[..., :, :] = rhs[..., :, :]in the least squares sense.

Below we will use the following notation for each pair of matrix and right-hand sides in the batch:

matrix=A∈ℜm×n, rhs=B∈ℜm×k, output=X∈ℜn×k, l2\_regularizer=λ.

If fast is True, then the solution is computed by solving the normal equations using Cholesky decomposition. Specifically, if m≥n then X=(ATA+λI)−1ATB, which solves the least-squares problem X=argminZ∈ℜn×k||AZ−B||F2+λ||Z||F2. If m<n then output is computed as X=AT(AAT+λI)−1B, which (for λ=0) is the minimum-norm solution to the under-determined linear system, i.e. X=argminZ∈ℜn×k||Z||F2, subject to AZ=B. Notice that the fast path is only numerically stable when A is numerically full rank and has a condition number cond(A)<1ϵmach orλ is sufficiently large.

If fast is False an algorithm based on the numerically robust complete orthogonal decomposition is used. This computes the minimum-norm least-squares solution, even when A is rank deficient. This path is typically 6-7 times slower than the fast path. If fast is False then l2\_regularizer is ignored.

#### Args:

* **matrix**: Tensor of shape [..., M, N].
* **rhs**: Tensor of shape [..., M, K].
* **l2\_regularizer**: 0-D double Tensor. Ignored if fast=False.
* **fast**: bool. Defaults to True.
* **name**: string, optional name of the operation.

#### Returns:

* **output**: Tensor of shape [..., N, K] whose inner-most 2 dimensions form M-by-Kmatrices that solve the equations matrix[..., :, :] \* output[..., :, :] = rhs[..., :, :] in the least squares sense.

#### Raises:

* **NotImplementedError**: linalg.lstsq is currently disabled for complex128 and l2\_regularizer != 0 due to poor accuracy.

# tf.linalg.lu

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/lu#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/lu#aliases)

Computes the LU decomposition of one or more square matrices.

### Aliases:

* tf.compat.v1.linalg.lu
* tf.compat.v2.linalg.lu
* tf.linalg.lu

tf.linalg.lu(  
    input,  
    output\_idx\_type=tf.dtypes.int32,  
    name=None  
)

Defined in generated file: python/ops/gen\_linalg\_ops.py.

The input is a tensor of shape [..., M, M] whose inner-most 2 dimensions form square matrices.

The input has to be invertible.

The output consists of two tensors LU and P containing the LU decomposition of all input submatrices [..., :, :]. LU encodes the lower triangular and upper triangular factors.

For each input submatrix of shape [M, M], L is a lower triangular matrix of shape [M, M] with unit diagonal whose entries correspond to the strictly lower triangular part of LU. U is a upper triangular matrix of shape [M, M] whose entries correspond to the upper triangular part, including the diagonal, of LU.

P represents a permutation matrix encoded as a list of indices each between 0 and M-1, inclusive. If P\_mat denotes the permutation matrix corresponding to P, then the L, U and P satisfies P\_mat \* input = L \* U.

#### Args:

* **input**: A Tensor. Must be one of the following types: float64, float32, half, complex64, complex128. A tensor of shape [..., M, M] whose inner-most 2 dimensions form matrices of size [M, M].
* **output\_idx\_type**: An optional [tf.DType](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/dtypes/DType) from: tf.int32, tf.int64. Defaults to [tf.int32](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf#int32).
* **name**: A name for the operation (optional).

#### Returns:

A tuple of Tensor objects (lu, p).

* **lu**: A Tensor. Has the same type as input.
* **p**: A Tensor of type output\_idx\_type.

# tf.linalg.matmul

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/matmul#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/matmul#aliases)
* [Used in the guide:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/matmul#used_in_the_guide)
* [Used in the tutorials:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/matmul#used_in_the_tutorials)

Multiplies matrix a by matrix b, producing a \* b.

### Aliases:

* tf.compat.v1.linalg.matmul
* tf.compat.v1.matmul
* tf.compat.v2.linalg.matmul
* tf.compat.v2.matmul
* tf.linalg.matmul
* tf.matmul

tf.linalg.matmul(  
    a,  
    b,  
    transpose\_a=False,  
    transpose\_b=False,  
    adjoint\_a=False,  
    adjoint\_b=False,  
    a\_is\_sparse=False,  
    b\_is\_sparse=False,  
    name=None  
)

Defined in [python/ops/math\_ops.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/math_ops.py).

### Used in the guide:

* [Eager essentials](https://www.tensorflow.org/beta/guide/eager)
* [Keras: A quick overview](https://www.tensorflow.org/beta/guide/keras/overview)
* [The Keras Functional API in TensorFlow](https://www.tensorflow.org/beta/guide/keras/functional)
* [Using GPUs](https://www.tensorflow.org/beta/guide/using_gpu)
* [Writing layers and models with TensorFlow Keras](https://www.tensorflow.org/beta/guide/keras/custom_layers_and_models)
* [tf.function and AutoGraph in TensorFlow 2.0](https://www.tensorflow.org/beta/guide/autograph)

### Used in the tutorials:

* [Custom layers](https://www.tensorflow.org/beta/tutorials/eager/custom_layers)
* [Tensors and Operations](https://www.tensorflow.org/beta/tutorials/eager/basics)
* [Transformer model for language understanding](https://www.tensorflow.org/beta/tutorials/text/transformer)
* [tf.function](https://www.tensorflow.org/beta/tutorials/eager/tf_function)

The inputs must, following any transpositions, be tensors of rank >= 2 where the inner 2 dimensions specify valid matrix multiplication arguments, and any further outer dimensions match.

Both matrices must be of the same type. The supported types are: float16, float32, float64, int32, complex64, complex128.

Either matrix can be transposed or adjointed (conjugated and transposed) on the fly by setting one of the corresponding flag to True. These are False by default.

If one or both of the matrices contain a lot of zeros, a more efficient multiplication algorithm can be used by setting the corresponding a\_is\_sparse or b\_is\_sparse flag to True. These are False by default. This optimization is only available for plain matrices (rank-2 tensors) with datatypes bfloat16 or float32.

#### For example:

# 2-D tensor `a`  
# [[1, 2, 3],  
#  [4, 5, 6]]  
a = tf.constant([1, 2, 3, 4, 5, 6], shape=[2, 3])  
  
# 2-D tensor `b`  
# [[ 7,  8],  
#  [ 9, 10],  
#  [11, 12]]  
b = tf.constant([7, 8, 9, 10, 11, 12], shape=[3, 2])  
  
# `a` \* `b`  
# [[ 58,  64],  
#  [139, 154]]  
c = tf.matmul(a, b)  
  
  
# 3-D tensor `a`  
# [[[ 1,  2,  3],  
#   [ 4,  5,  6]],  
#  [[ 7,  8,  9],  
#   [10, 11, 12]]]  
a = tf.constant(np.arange(1, 13, dtype=np.int32),  
                shape=[2, 2, 3])  
  
# 3-D tensor `b`  
# [[[13, 14],  
#   [15, 16],  
#   [17, 18]],  
#  [[19, 20],  
#   [21, 22],  
#   [23, 24]]]  
b = tf.constant(np.arange(13, 25, dtype=np.int32),  
                shape=[2, 3, 2])  
  
# `a` \* `b`  
# [[[ 94, 100],  
#   [229, 244]],  
#  [[508, 532],  
#   [697, 730]]]  
c = tf.matmul(a, b)  
  
# Since python >= 3.5 the @ operator is supported (see PEP 465).  
# In TensorFlow, it simply calls the `tf.matmul()` function, so the  
# following lines are equivalent:  
d = a @ b @ [[10.], [11.]]  
d = tf.matmul(tf.matmul(a, b), [[10.], [11.]])

#### Args:

* **a**: Tensor of type float16, float32, float64, int32, complex64, complex128 and rank > 1.
* **b**: Tensor with same type and rank as a.
* **transpose\_a**: If True, a is transposed before multiplication.
* **transpose\_b**: If True, b is transposed before multiplication.
* **adjoint\_a**: If True, a is conjugated and transposed before multiplication.
* **adjoint\_b**: If True, b is conjugated and transposed before multiplication.
* **a\_is\_sparse**: If True, a is treated as a sparse matrix.
* **b\_is\_sparse**: If True, b is treated as a sparse matrix.
* **name**: Name for the operation (optional).

#### Returns:

A Tensor of the same type as a and b where each inner-most matrix is the product of the corresponding matrices in a and b, e.g. if all transpose or adjoint attributes are False:

output[..., i, j] = sum\_k (a[..., i, k] \* b[..., k, j]), for all indices i, j.

* **Note**: This is matrix product, not element-wise product.

#### Raises:

* **ValueError**: If transpose\_a and adjoint\_a, or transpose\_b and adjoint\_b are both set to True.

# tf.linalg.matrix\_transpose

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/matrix_transpose#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/matrix_transpose#aliases)

Transposes last two dimensions of tensor a.

### Aliases:

* tf.compat.v1.linalg.matrix\_transpose
* tf.compat.v1.linalg.transpose
* tf.compat.v1.matrix\_transpose
* tf.compat.v2.linalg.matrix\_transpose
* tf.linalg.matrix\_transpose

tf.linalg.matrix\_transpose(  
    a,  
    name='matrix\_transpose',  
    conjugate=False  
)

Defined in [python/ops/array\_ops.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/array_ops.py).

#### For example:

x = tf.constant([[1, 2, 3], [4, 5, 6]])  
tf.linalg.matrix\_transpose(x)  # [[1, 4],  
                               #  [2, 5],  
                               #  [3, 6]]  
  
x = tf.constant([[1 + 1j, 2 + 2j, 3 + 3j],  
                 [4 + 4j, 5 + 5j, 6 + 6j]])  
tf.linalg.matrix\_transpose(x, conjugate=True)  # [[1 - 1j, 4 - 4j],  
                                               #  [2 - 2j, 5 - 5j],  
                                               #  [3 - 3j, 6 - 6j]]  
  
# Matrix with two batch dimensions.  
# x.shape is [1, 2, 3, 4]  
# tf.linalg.matrix\_transpose(x) is shape [1, 2, 4, 3]

Note that [tf.matmul](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/matmul) provides kwargs allowing for transpose of arguments. This is done with minimal cost, and is preferable to using this function. E.g.

# Good!  Transpose is taken at minimal additional cost.  
tf.matmul(matrix, b, transpose\_b=True)  
  
# Inefficient!  
tf.matmul(matrix, tf.linalg.matrix\_transpose(b))

#### Args:

* **a**: A Tensor with rank >= 2.
* **name**: A name for the operation (optional).
* **conjugate**: Optional bool. Setting it to True is mathematically equivalent to tf.math.conj(tf.linalg.matrix\_transpose(input)).

#### Returns:

A transposed batch matrix Tensor.

#### Raises:

* **ValueError**: If a is determined statically to have rank < 2.

#### Numpy Compatibility

In numpy transposes are memory-efficient constant time operations as they simply return a new view of the same data with adjusted strides.

TensorFlow does not support strides, linalg.matrix\_transpose returns a new tensor with the items permuted.

# tf.linalg.matvec

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/matvec#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/matvec#aliases)

Multiplies matrix a by vector b, producing a \* b.

### Aliases:

* tf.compat.v1.linalg.matvec
* tf.compat.v2.linalg.matvec
* tf.linalg.matvec

tf.linalg.matvec(  
    a,  
    b,  
    transpose\_a=False,  
    adjoint\_a=False,  
    a\_is\_sparse=False,  
    b\_is\_sparse=False,  
    name=None  
)

Defined in [python/ops/math\_ops.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/math_ops.py).

The matrix a must, following any transpositions, be a tensor of rank >= 2, and we must have shape(b) = shape(a)[:-2] + [shape(a)[-1]].

Both a and b must be of the same type. The supported types are: float16, float32, float64, int32, complex64, complex128.

Matrix a can be transposed or adjointed (conjugated and transposed) on the fly by setting one of the corresponding flag to True. These are False by default.

If one or both of the inputs contain a lot of zeros, a more efficient multiplication algorithm can be used by setting the corresponding a\_is\_sparse or b\_is\_sparse flag to True. These are False by default. This optimization is only available for plain matrices/vectors (rank-2/1 tensors) with datatypes bfloat16 or float32.

#### For example:

# 2-D tensor `a`  
# [[1, 2, 3],  
#  [4, 5, 6]]  
a = tf.constant([1, 2, 3, 4, 5, 6], shape=[2, 3])  
  
# 1-D tensor `b`  
# [7, 9, 11]  
b = tf.constant([7, 9, 11], shape=[3])  
  
# `a` \* `b`  
# [ 58,  64]  
c = tf.matvec(a, b)  
  
  
# 3-D tensor `a`  
# [[[ 1,  2,  3],  
#   [ 4,  5,  6]],  
#  [[ 7,  8,  9],  
#   [10, 11, 12]]]  
a = tf.constant(np.arange(1, 13, dtype=np.int32),  
                shape=[2, 2, 3])  
  
# 2-D tensor `b`  
# [[13, 14, 15],  
#  [16, 17, 18]]  
b = tf.constant(np.arange(13, 19, dtype=np.int32),  
                shape=[2, 3])  
  
# `a` \* `b`  
# [[ 86, 212],  
#  [410, 563]]  
c = tf.matvec(a, b)

#### Args:

* **a**: Tensor of type float16, float32, float64, int32, complex64, complex128 and rank > 1.
* **b**: Tensor with same type and rank = rank(a) - 1.
* **transpose\_a**: If True, a is transposed before multiplication.
* **adjoint\_a**: If True, a is conjugated and transposed before multiplication.
* **a\_is\_sparse**: If True, a is treated as a sparse matrix.
* **b\_is\_sparse**: If True, b is treated as a sparse matrix.
* **name**: Name for the operation (optional).

#### Returns:

A Tensor of the same type as a and b where each inner-most vector is the product of the corresponding matrices in a and vectors in b, e.g. if all transpose or adjoint attributes are False:

output[..., i] = sum\_k (a[..., i, k] \* b[..., k]), for all indices i.

* **Note**: This is matrix-vector product, not element-wise product.

#### Raises:

* **ValueError**: If transpose\_a and adjoint\_a are both set to True.

# tf.linalg.qr

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/qr#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/qr#aliases)

Computes the QR decompositions of one or more matrices.

### Aliases:

* tf.compat.v1.linalg.qr
* tf.compat.v1.qr
* tf.compat.v2.linalg.qr
* tf.linalg.qr

tf.linalg.qr(  
    input,  
    full\_matrices=False,  
    name=None  
)

Defined in generated file: python/ops/gen\_linalg\_ops.py.

Computes the QR decomposition of each inner matrix in tensor such that tensor[..., :, :] = q[..., :, :] \* r[..., :,:])

# a is a tensor.  
# q is a tensor of orthonormal matrices.  
# r is a tensor of upper triangular matrices.  
q, r = qr(a)  
q\_full, r\_full = qr(a, full\_matrices=True)

#### Args:

* **input**: A Tensor. Must be one of the following types: float64, float32, half, complex64, complex128. A tensor of shape [..., M, N] whose inner-most 2 dimensions form matrices of size [M, N]. Let P be the minimum of M and N.
* **full\_matrices**: An optional bool. Defaults to False. If true, compute full-sized q and r. If false (the default), compute only the leading P columns of q.
* **name**: A name for the operation (optional).

#### Returns:

A tuple of Tensor objects (q, r).

* **q**: A Tensor. Has the same type as input.
* **r**: A Tensor. Has the same type as input.

# tf.linalg.set\_diag

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/set_diag#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/set_diag#aliases)

Returns a batched matrix tensor with new batched diagonal values.

### Aliases:

* tf.compat.v1.linalg.set\_diag
* tf.compat.v1.matrix\_set\_diag
* tf.compat.v2.linalg.set\_diag
* tf.linalg.set\_diag

tf.linalg.set\_diag(  
    input,  
    diagonal,  
    name=None  
)

Defined in generated file: python/ops/gen\_array\_ops.py.

Given input and diagonal, this operation returns a tensor with the same shape and values as input, except for the main diagonal of the innermost matrices. These will be overwritten by the values in diagonal.

The output is computed as follows:

Assume input has k+1 dimensions [I, J, K, ..., M, N] and diagonal has k dimensions [I, J, K, ..., min(M, N)]. Then the output is a tensor of rank k+1 with dimensions [I, J, K, ..., M, N] where:

* output[i, j, k, ..., m, n] = diagonal[i, j, k, ..., n] for m == n.
* output[i, j, k, ..., m, n] = input[i, j, k, ..., m, n] for m != n.

#### Args:

* **input**: A Tensor. Rank k+1, where k >= 1.
* **diagonal**: A Tensor. Must have the same type as input. Rank k, where k >= 1.
* **name**: A name for the operation (optional).

#### Returns:

A Tensor. Has the same type as input.

# tf.linalg.slogdet

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/slogdet#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/slogdet#aliases)

Computes the sign and the log of the absolute value of the determinant of

### Aliases:

* tf.compat.v1.linalg.slogdet
* tf.compat.v2.linalg.slogdet
* tf.linalg.slogdet

tf.linalg.slogdet(  
    input,  
    name=None  
)

Defined in generated file: python/ops/gen\_linalg\_ops.py.

one or more square matrices.

The input is a tensor of shape [N, M, M] whose inner-most 2 dimensions form square matrices. The outputs are two tensors containing the signs and absolute values of the log determinants for all N input submatrices [..., :, :] such that the determinant = signexp(log\_abs\_determinant). The log\_abs\_determinant is computed as det(P)sum(log(diag(LU))) where LU is the LU decomposition of the input and P is the corresponding permutation matrix.

#### Args:

* **input**: A Tensor. Must be one of the following types: half, float32, float64, complex64, complex128. Shape is [N, M, M].
* **name**: A name for the operation (optional).

#### Returns:

A tuple of Tensor objects (sign, log\_abs\_determinant).

* **sign**: A Tensor. Has the same type as input.
* **log\_abs\_determinant**: A Tensor. Has the same type as input.

# tf.linalg.solve

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/solve#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/solve#aliases)

Solves systems of linear equations.

### Aliases:

* tf.compat.v1.linalg.solve
* tf.compat.v1.matrix\_solve
* tf.compat.v2.linalg.solve
* tf.linalg.solve

tf.linalg.solve(  
    matrix,  
    rhs,  
    adjoint=False,  
    name=None  
)

Defined in generated file: python/ops/gen\_linalg\_ops.py.

Matrix is a tensor of shape [..., M, M] whose inner-most 2 dimensions form square matrices. Rhs is a tensor of shape [..., M, K]. The output is a tensor shape [..., M, K]. If adjoint is False then each output matrix satisfies matrix[..., :, :] \* output[..., :, :] = rhs[..., :, :]. If adjoint is True then each output matrix satisfies adjoint(matrix[..., :, :]) \* output[..., :, :] = rhs[..., :, :].

#### Args:

* **matrix**: A Tensor. Must be one of the following types: float64, float32, half, complex64, complex128. Shape is [..., M, M].
* **rhs**: A Tensor. Must have the same type as matrix. Shape is [..., M, K].
* **adjoint**: An optional bool. Defaults to False. Boolean indicating whether to solve with matrix or its (block-wise) adjoint.
* **name**: A name for the operation (optional).

#### Returns:

A Tensor. Has the same type as matrix.

# tf.linalg.sqrtm

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/sqrtm#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/sqrtm#aliases)

Computes the matrix square root of one or more square matrices:

### Aliases:

* tf.compat.v1.linalg.sqrtm
* tf.compat.v1.matrix\_square\_root
* tf.compat.v2.linalg.sqrtm
* tf.compat.v2.matrix\_square\_root
* tf.linalg.sqrtm
* tf.matrix\_square\_root

tf.linalg.sqrtm(  
    input,  
    name=None  
)

Defined in generated file: python/ops/gen\_linalg\_ops.py.

matmul(sqrtm(A), sqrtm(A)) = A

The input matrix should be invertible. If the input matrix is real, it should have no eigenvalues which are real and negative (pairs of complex conjugate eigenvalues are allowed).

The matrix square root is computed by first reducing the matrix to quasi-triangular form with the real Schur decomposition. The square root of the quasi-triangular matrix is then computed directly. Details of the algorithm can be found in: Nicholas J. Higham, "Computing real square roots of a real matrix", Linear Algebra Appl., 1987.

The input is a tensor of shape [..., M, M] whose inner-most 2 dimensions form square matrices. The output is a tensor of the same shape as the input containing the matrix square root for all input submatrices [..., :, :].

#### Args:

* **input**: A Tensor. Must be one of the following types: float64, float32, half, complex64, complex128. Shape is [..., M, M].
* **name**: A name for the operation (optional).

#### Returns:

A Tensor. Has the same type as input.

# tf.linalg.svd

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/svd#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/svd#aliases)

Computes the singular value decompositions of one or more matrices.

### Aliases:

* tf.compat.v1.linalg.svd
* tf.compat.v1.svd
* tf.compat.v2.linalg.svd
* tf.linalg.svd

tf.linalg.svd(  
    tensor,  
    full\_matrices=False,  
    compute\_uv=True,  
    name=None  
)

Defined in [python/ops/linalg\_ops.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg_ops.py).

Computes the SVD of each inner matrix in tensor such that tensor[..., :, :] = u[..., :, :] \* diag(s[..., :, :]) \* transpose(conj(v[..., :, :]))

# a is a tensor.  
# s is a tensor of singular values.  
# u is a tensor of left singular vectors.  
# v is a tensor of right singular vectors.  
s, u, v = svd(a)  
s = svd(a, compute\_uv=False)

#### Args:

* **tensor**: Tensor of shape [..., M, N]. Let P be the minimum of M and N.
* **full\_matrices**: If true, compute full-sized u and v. If false (the default), compute only the leading P singular vectors. Ignored if compute\_uv is False.
* **compute\_uv**: If True then left and right singular vectors will be computed and returned in uand v, respectively. Otherwise, only the singular values will be computed, which can be significantly faster.
* **name**: string, optional name of the operation.

#### Returns:

* **s**: Singular values. Shape is [..., P]. The values are sorted in reverse order of magnitude, so s[..., 0] is the largest value, s[..., 1] is the second largest, etc.
* **u**: Left singular vectors. If full\_matrices is False (default) then shape is [..., M, P]; if full\_matrices is True then shape is [..., M, M]. Not returned if compute\_uv is False.
* **v**: Right singular vectors. If full\_matrices is False (default) then shape is [..., N, P]. If full\_matrices is True then shape is [..., N, N]. Not returned if compute\_uv is False.

#### Numpy Compatibility

Mostly equivalent to numpy.linalg.svd, except that \* The order of output arguments here is s, u, vwhen compute\_uv is True, as opposed to u, s, v for numpy.linalg.svd. \* full\_matrices is False by default as opposed to True for numpy.linalg.svd. \* tf.linalg.svd uses the standard definition of the SVD A=UΣVH, such that the left singular vectors of a are the columns of u, while the right singular vectors of a are the columns of v. On the other hand, numpy.linalg.svd returns the adjointVH as the third output argument.

import tensorflow as tf  
import numpy as np  
s, u, v = tf.linalg.svd(a)  
tf\_a\_approx = tf.matmul(u, tf.matmul(tf.linalg.diag(s), v, adjoint\_b=True))  
u, s, v\_adj = np.linalg.svd(a, full\_matrices=False)  
np\_a\_approx = np.dot(u, np.dot(np.diag(s), v\_adj))  
# tf\_a\_approx and np\_a\_approx should be numerically close.

# tf.linalg.tensor\_diag

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/tensor_diag#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/tensor_diag#aliases)

Returns a diagonal tensor with a given diagonal values.

### Aliases:

* tf.compat.v1.diag
* tf.compat.v1.linalg.tensor\_diag
* tf.compat.v2.linalg.tensor\_diag
* tf.linalg.tensor\_diag

tf.linalg.tensor\_diag(  
    diagonal,  
    name=None  
)

Defined in generated file: python/ops/gen\_array\_ops.py.

Given a diagonal, this operation returns a tensor with the diagonal and everything else padded with zeros. The diagonal is computed as follows:

Assume diagonal has dimensions [D1,..., Dk], then the output is a tensor of rank 2k with dimensions [D1,..., Dk, D1,..., Dk] where:

output[i1,..., ik, i1,..., ik] = diagonal[i1, ..., ik] and 0 everywhere else.

#### For example:

# 'diagonal' is [1, 2, 3, 4]  
tf.diag(diagonal) ==> [[1, 0, 0, 0]  
                       [0, 2, 0, 0]  
                       [0, 0, 3, 0]  
                       [0, 0, 0, 4]]

#### Args:

* **diagonal**: A Tensor. Must be one of the following types: bfloat16, half, float32, float64, int32, int64, complex64, complex128. Rank k tensor where k is at most 1.
* **name**: A name for the operation (optional).

#### Returns:

A Tensor. Has the same type as diagonal.

# tf.linalg.tensor\_diag\_part

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/tensor_diag_part#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/tensor_diag_part#aliases)

Returns the diagonal part of the tensor.

### Aliases:

* tf.compat.v1.diag\_part
* tf.compat.v1.linalg.tensor\_diag\_part
* tf.compat.v2.linalg.tensor\_diag\_part
* tf.linalg.tensor\_diag\_part

tf.linalg.tensor\_diag\_part(  
    input,  
    name=None  
)

Defined in generated file: python/ops/gen\_array\_ops.py.

This operation returns a tensor with the diagonal part of the input. The diagonal part is computed as follows:

Assume input has dimensions [D1,..., Dk, D1,..., Dk], then the output is a tensor of rank kwith dimensions [D1,..., Dk] where:

diagonal[i1,..., ik] = input[i1, ..., ik, i1,..., ik].

#### For example:

# 'input' is [[1, 0, 0, 0]  
              [0, 2, 0, 0]  
              [0, 0, 3, 0]  
              [0, 0, 0, 4]]  
  
tf.diag\_part(input) ==> [1, 2, 3, 4]

#### Args:

* **input**: A Tensor. Must be one of the following types: bfloat16, half, float32, float64, int32, int64, complex64, complex128. Rank k tensor where k is even and not zero.
* **name**: A name for the operation (optional).

#### Returns:

A Tensor. Has the same type as input.

# tf.linalg.trace

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/trace#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/trace#aliases)

Compute the trace of a tensor x.

### Aliases:

* tf.compat.v1.linalg.trace
* tf.compat.v1.trace
* tf.compat.v2.linalg.trace
* tf.linalg.trace

tf.linalg.trace(  
    x,  
    name=None  
)

Defined in [python/ops/math\_ops.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/math_ops.py).

trace(x) returns the sum along the main diagonal of each inner-most matrix in x. If x is of rank kwith shape [I, J, K, ..., L, M, N], then output is a tensor of rank k-2 with dimensions [I, J, K, ..., L] where

output[i, j, k, ..., l] = trace(x[i, j, i, ..., l, :, :])

#### For example:

x = tf.constant([[1, 2], [3, 4]])  
tf.linalg.trace(x)  # 5  
  
x = tf.constant([[1, 2, 3],  
                 [4, 5, 6],  
                 [7, 8, 9]])  
tf.linalg.trace(x)  # 15  
  
x = tf.constant([[[1, 2, 3],  
                  [4, 5, 6],  
                  [7, 8, 9]],  
                 [[-1, -2, -3],  
                  [-4, -5, -6],  
                  [-7, -8, -9]]])  
tf.linalg.trace(x)  # [15, -15]

#### Args:

* **x**: tensor.
* **name**: A name for the operation (optional).

#### Returns:

The trace of input tensor.

# tf.linalg.triangular\_solve

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/triangular_solve#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/triangular_solve#aliases)

Solves systems of linear equations with upper or lower triangular matrices by backsubstitution.

### Aliases:

* tf.compat.v1.linalg.triangular\_solve
* tf.compat.v1.matrix\_triangular\_solve
* tf.compat.v2.linalg.triangular\_solve
* tf.linalg.triangular\_solve

tf.linalg.triangular\_solve(  
    matrix,  
    rhs,  
    lower=True,  
    adjoint=False,  
    name=None  
)

Defined in generated file: python/ops/gen\_linalg\_ops.py.

matrix is a tensor of shape [..., M, M] whose inner-most 2 dimensions form square matrices. If lower is True then the strictly upper triangular part of each inner-most matrix is assumed to be zero and not accessed. If lower is False then the strictly lower triangular part of each inner-most matrix is assumed to be zero and not accessed. rhs is a tensor of shape [..., M, K].

The output is a tensor of shape [..., M, K]. If adjoint is True then the innermost matrices in output satisfy matrix equations matrix[..., :, :] \* output[..., :, :] = rhs[..., :, :]. If adjoint is False then the strictly then the innermost matrices in output satisfy matrix equationsadjoint(matrix[..., i, k]) \* output[..., k, j] = rhs[..., i, j].

#### Example:

a = tf.constant([[3,  0,  0,  0],  
                 [2,  1,  0,  0],  
                 [1,  0,  1,  0],  
                 [1,  1,  1,  1]], dtype=tf.float32)  
  
b = tf.constant([[4],  
                 [2],  
                 [4],  
                 [2]], dtype=tf.float32)  
  
x = tf.linalg.triangular\_solve(a, b, lower=True)  
x  
# <tf.Tensor: id=257, shape=(4, 1), dtype=float32, numpy=  
# array([[ 1.3333334 ],  
#        [-0.66666675],  
#        [ 2.6666665 ],  
#        [-1.3333331 ]], dtype=float32)>  
  
# in python3 one can use `a@x`  
tf.matmul(a, x)  
# <tf.Tensor: id=263, shape=(4, 1), dtype=float32, numpy=  
# array([[4.       ],  
#        [2.       ],  
#        [4.       ],  
#        [1.9999999]], dtype=float32)>

#### Args:

* **matrix**: A Tensor. Must be one of the following types: float64, float32, half, complex64, complex128. Shape is [..., M, M].
* **rhs**: A Tensor. Must have the same type as matrix. Shape is [..., M, K].
* **lower**: An optional bool. Defaults to True. Boolean indicating whether the innermost matrices in matrix are lower or upper triangular.
* **adjoint**: An optional bool. Defaults to False. Boolean indicating whether to solve with matrix or its (block-wise) adjoint.
* **name**: A name for the operation (optional).

#### Returns:

A Tensor. Has the same type as matrix.

#### Numpy Compatibility

Equivalent to scipy.linalg.solve\_triangular

# tf.linalg.tridiagonal\_matmul

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/tridiagonal_matmul#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/tridiagonal_matmul#aliases)

Multiplies tridiagonal matrix by matrix.

### Aliases:

* tf.compat.v1.linalg.tridiagonal\_matmul
* tf.compat.v2.linalg.tridiagonal\_matmul
* tf.linalg.tridiagonal\_matmul

tf.linalg.tridiagonal\_matmul(  
    diagonals,  
    rhs,  
    diagonals\_format='compact',  
    name=None  
)

Defined in [python/ops/linalg/linalg\_impl.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg/linalg_impl.py).

diagonals is representation of 3-diagonal NxN matrix, which depends on diagonals\_format.

In matrix format, diagonals must be a tensor of shape [..., M, M], with two inner-most dimensions representing the square tridiagonal matrices. Elements outside of the three diagonals will be ignored.

If sequence format, diagonals is list or tuple of three tensors: [superdiag, maindiag, subdiag], each having shape [..., M]. Last element of superdiag first element of subdiag are ignored.

In compact format the three diagonals are brought together into one tensor of shape [..., 3, M], with last two dimensions containing superdiagonals, diagonals, and subdiagonals, in order. Similarly to sequence format, elements diagonals[..., 0, M-1] and diagonals[..., 2, 0] are ignored.

The sequence format is recommended as the one with the best performance.

rhs is matrix to the right of multiplication. It has shape [..., M, N].

#### Example:

superdiag = tf.constant([-1, -1, 0], dtype=tf.float64)  
maindiag = tf.constant([2, 2, 2], dtype=tf.float64)  
subdiag = tf.constant([0, -1, -1], dtype=tf.float64)  
diagonals = [superdiag, maindiag, subdiag]  
rhs = tf.constant([[1, 1], [1, 1], [1, 1]], dtype=tf.float64)  
x = tf.linalg.tridiagonal\_matmul(diagonals, rhs, diagonals\_format='sequence')

#### Args:

* **diagonals**: A Tensor or tuple of Tensors describing left-hand sides. The shape depends of diagonals\_format, see description above. Must be float32, float64, complex64, or complex128.
* **rhs**: A Tensor of shape [..., M, N] and with the same dtype as diagonals.
* **diagonals\_format**: one of sequence, or compact. Default is compact.
* **name**: A name to give this Op (optional).

#### Returns:

A Tensor of shape [..., M, N] containing the result of multiplication.

#### Raises:

* **ValueError**: An unsupported type is provided as input, or when the input tensors have incorrect shapes.

# tf.linalg.tridiagonal\_solve

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/tridiagonal_solve#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/linalg/tridiagonal_solve#aliases)

Solves tridiagonal systems of equations.

### Aliases:

* tf.compat.v1.linalg.tridiagonal\_solve
* tf.compat.v2.linalg.tridiagonal\_solve
* tf.linalg.tridiagonal\_solve

tf.linalg.tridiagonal\_solve(  
    diagonals,  
    rhs,  
    diagonals\_format='compact',  
    transpose\_rhs=False,  
    conjugate\_rhs=False,  
    name=None,  
    partial\_pivoting=True  
)

Defined in [python/ops/linalg/linalg\_impl.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/ops/linalg/linalg_impl.py).

The input can be supplied in various formats: matrix, sequence and compact, specified by the diagonals\_format arg.

In matrix format, diagonals must be a tensor of shape [..., M, M], with two inner-most dimensions representing the square tridiagonal matrices. Elements outside of the three diagonals will be ignored.

In sequence format, diagonals are supplied as a tuple or list of three tensors of shapes [..., N], [..., M], [..., N] representing superdiagonals, diagonals, and subdiagonals, respectively. N can be either M-1 or M; in the latter case, the last element of superdiagonal and the first element of subdiagonal will be ignored.

In compact format the three diagonals are brought together into one tensor of shape [..., 3, M], with last two dimensions containing superdiagonals, diagonals, and subdiagonals, in order. Similarly to sequence format, elements diagonals[..., 0, M-1] and diagonals[..., 2, 0] are ignored.

The compact format is recommended as the one with best performance. In case you need to cast a tensor into a compact format manually, use [tf.gather\_nd](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/gather_nd). An example for a tensor of shape [m, m]:

rhs = tf.constant([...])  
matrix = tf.constant([[...]])  
m = matrix.shape[0]  
dummy\_idx = [0, 0]  # An arbitrary element to use as a dummy  
indices = [[[i, i + 1] for i in range(m - 1)] + [dummy\_idx],  # Superdiagonal  
         [[i, i] for i in range(m)],                          # Diagonal  
         [dummy\_idx] + [[i + 1, i] for i in range(m - 1)]]    # Subdiagonal  
diagonals=tf.gather\_nd(matrix, indices)  
x = tf.linalg.tridiagonal\_solve(diagonals, rhs)

Regardless of the diagonals\_format, rhs is a tensor of shape [..., M] or [..., M, K]. The latter allows to simultaneously solve K systems with the same left-hand sides and K different right-hand sides. If transpose\_rhs is set to True the expected shape is [..., M] or [..., K, M].

The batch dimensions, denoted as ..., must be the same in diagonals and rhs.

The output is a tensor of the same shape as rhs: either [..., M] or [..., M, K].

The op isn't guaranteed to raise an error if the input matrix is not invertible. [tf.debugging.check\_numerics](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/debugging/check_numerics) can be applied to the output to detect invertibility problems.

**Note**: with large batch sizes, the computation on the GPU may be slow, if either partial\_pivoting=True or there are multiple right-hand sides (K > 1). If this issue arises, consider if it's possible to disable pivoting and have K = 1, or, alternatively, consider using CPU.

On CPU, solution is computed via Gaussian elimination with or without partial pivoting, depending on partial\_pivoting parameter. On GPU, Nvidia's cuSPARSE library is used: https://docs.nvidia.com/cuda/cusparse/index.html#gtsv

#### Args:

* **diagonals**: A Tensor or tuple of Tensors describing left-hand sides. The shape depends of diagonals\_format, see description above. Must be float32, float64, complex64, or complex128.
* **rhs**: A Tensor of shape [..., M] or [..., M, K] and with the same dtype as diagonals.
* **diagonals\_format**: one of matrix, sequence, or compact. Default is compact.
* **transpose\_rhs**: If True, rhs is transposed before solving (has no effect if the shape of rhs is [..., M]).
* **conjugate\_rhs**: If True, rhs is conjugated before solving.
* **name**: A name to give this Op (optional).
* **partial\_pivoting**: whether to perform partial pivoting. True by default. Partial pivoting makes the procedure more stable, but slower. Partial pivoting is unnecessary in some cases, including diagonally dominant and symmetric positive definite matrices (see e.g. theorem 9.12 in [1]).

#### Returns:

A Tensor of shape [..., M] or [..., M, K] containing the solutions.

#### Raises:

* **ValueError**: An unsupported type is provided as input, or when the input tensors have incorrect shapes.

[1] Nicholas J. Higham (2002). Accuracy and Stability of Numerical Algorithms: Second Edition. SIAM. p. 175. ISBN 978-0-89871-802-7.