

# 1 Appendix

## 1.A linalg.dot

```
def test_dot_corner(self):  
    actual = np.dot([ ], [ ])   
    expected = False  
    self.assertEqual(actual, expected);
```

```
def test_dot_corner2(self):  
    with self.assertRaises(ValueError):  
        actual = np.dot([ ], [1, 2])
```

```
def test_dot_iden(self):  
    actual = np.dot(self.array_1, self.array_2)  
    expected = [[4, 1], [2, 2]]  
    self.assertTrue((actual == expected).all())
```

```
def test_dot_zero(self):  
    actual = np.dot(self.array_zero, self.array_2)  
    expected = 0  
    self.assertTrue((actual == expected).all())
```

```
def test_dot_com(self):  
    actual = np.dot(self.array_com_1, self.array_com_2)  
    expected = np.dot(self.array_com_2, self.array_com_1)  
    self.assertTrue((actual == expected).all())
```

```
def test_dot_square(self):  
    actual = np.dot(self.array_2, self.array_2)  
    expected = [[18, 6], [12, 6]]  
    self.assertTrue((actual == expected).all())
```

```
def test_dot_raises(self):  
    with self.assertRaises(ValueError):  
        actual = np.dot([2, 2, 3], [2, 1])
```

## 1.B linalg.vdot

```
def test_vdot_square(self):
    actual = np.vdot(self.array_a, self.array_a)
    expected = 78
    self.assertTrue(actual == expected)
```

```
def test_vdot_matrix(self):
    actual = np.vdot(self.array_a, self.array_a)
    expected = 78
    self.assertTrue(actual == expected)
```

```
def test_vdot_normal(self):
    self.setupForComplex()
    actual = np.vdot(self.complex_a, self.complex_b)
    expected = 70-8j
    self.assertTrue(actual == expected)
```

```
def test_vdot_com(self):
    self.setupForComplex()
    actual = np.vdot(self.array_a, self.array_b)
    expected = np.vdot(self.array_b, self.array_a)
    self.assertTrue(actual == expected)
```

```
def test_vdot_negative(self):
    self.setupForComplex()
    actual = np.vdot(self.complex_c, self.complex_a)
    expected = -70-8j
    self.assertTrue(actual == expected)
```

```
def test_vdot_float(self):
    actual = np.vdot(self.array_a_float, self.array_b_float)
    expected = 14.25
    self.assertTrue(actual == expected)
```

```
def test_vdot_empty(self):
    actual = np.vdot([],[])
    self.assertFalse(actual)
```

## 1.C linalg.multi\_dot

```
def test_dot_raises(self):
    with self.assertRaises(ValueError):
        actual = np.dot([2, 2, 3], [2, 1])
```

Figure 1: Raising exception in dot product.

## 2 Black-box Testing

### 2.A linalg.dot

#### 2.A.1 documentation

For 2-D arrays it is equivalent to matrix multiplication, and for 1-D arrays to inner product of vectors (without complex conjugation). For N dimensions it is a sum product over the last axis of a and the second-to-last of b:

$$\text{dot}(a, b)[i,j,k,m] = \text{sum}(a[i,j,:] * b[k,:,m])$$

**Parameters** : two arrays a: array\_like First argument.

b : array\_like Second argument.

out : ndarray, optional output argument. This must have the exact kind that would be returned if it was not used. In particular, it must have the right type, must be C-contiguous, and its dtype must be the dtype that would be returned for dot(a,b). This is a performance feature. Therefore, if these conditions are not met, an exception is raised, instead of attempting to be flexible.

**Returns** : output : ndarray

Returns the dot product of a and b. If a and b are both scalars or both 1-D arrays then a scalar is returned; otherwise an array is returned. If out is given, then it is returned. Raises: ValueError If the last dimension of a is not the same size as the second-to-last dimension of b.

#### 2.A.2 tests

### 2.B linalg.multidot

Compute the dot product of two or more arrays in a single function call, while automatically selecting the fastest evaluation order.

multi\_dot chains numpy.dot and uses optimal parenthesization of the matrices [R44] [R45]. Depending on the shapes of the matrices, this can speed up the multiplication a lot.

If the first argument is 1-D it is treated as a row vector. If the last argument is 1-D it is treated as a column vector. The other arguments must be 2-D.

#### 2.B.1 tests

### 2.C linalg.vdot

#### 2.C.1 documentation

Return the dot product of two vectors.

The `vdot(a, b)` function handles complex numbers differently than `dot(a, b)`. If the first argument is complex the complex conjugate of the first argument is used for the calculation of the dot product.

Note that `vdot` handles multidimensional arrays differently than `dot`: it does not perform a matrix product, but flattens input arguments to 1-D vectors first. Consequently, it should only be used for vectors.

## 2.C.2 tests

## 2.D `linalg.inner`

### 2.D.1 documentation

Inner product of two arrays.

Ordinary inner product of vectors for 1-D arrays (without complex conjugation), in higher dimensions a sum product over the last axes.

### 2.D.2 tests

## 2.E `linalg.outer`

### 2.E.1 documentation

Compute the outer product of two vectors.

**Parameters** : `a` : (M,) array\_like First input vector. Input is flattened if not already 1-dimensional. `b` : (N,) array\_like Second input vector. Input is flattened if not already 1-dimensional.

`out` : (M, N) ndarray, optional A location where the result is stored

**Returns** : `out` : (M, N) ndarray  $out[i, j] = a[i] * b[j]$

### 2.E.2 tests

## 2.F `linalg.matmul`

### 2.F.1 documentation

Matrix product of two arrays.

The behavior depends on the arguments in the following way.

If both arguments are 2-D they are multiplied like conventional matrices.

If either argument is N-D,  $N \geq 2$ , it is treated as a stack of matrices residing in the last two indexes and broadcast accordingly.

If the first argument is 1-D, it is promoted to a matrix by prepending a 1 to its dimensions. After matrix multiplication the prepended 1 is removed.

If the second argument is 1-D, it is promoted to a matrix by appending a 1 to its dimensions. After matrix multiplication the appended 1 is removed.

Multiplication by a scalar is not allowed, use `*` instead. Note that multiplying a stack of matrices with a vector will result in a stack of vectors, but `matmul` will not recognize it as such.

## **2.F.2 tests**

## **2.G linalg.tensor\_dot**

### **2.G.1 documentation**

Compute tensor dot product along specified axes for arrays  $i=1$ -D.

Given two tensors (arrays of dimension greater than or equal to one), a and b, and an array\_like object containing two array\_like objects, (a\_axes, b\_axes), sum the products of as and bs elements (components) over the axes specified by a\_axes and b\_axes. The third argument can be a single non-negative integer-like scalar, N; if it is such, then the last N dimensions of a and the first N dimensions of b are summed over.

### **2.G.2 tests**

## **2.H linalg.matrix\_power**

### **2.H.1 documentation**

Raise a square matrix to the (integer) power n.

For positive integers n, the power is computed by repeated matrix squarings and matrix multiplications. If n == 0, the identity matrix of the same shape as M is returned. If n  $\neq$  0, the inverse is computed and then raised to the abs(n).

### **2.H.2 tests**

## **2.I linalg.eig**

Compute the eigenvalues and right eigenvectors of a square array.

**Parameters** : a : (... , M, M) array

**Returns** : w : (... , M) array

Raises: LinAlgError If the eigenvalue computation does not converge.

### **2.I.1 documentation**

### **2.I.2 tests**

## **2.J linalg.eigh**

### **2.J.1 documentation**

Return the eigenvalues and eigenvectors of a Hermitian or symmetric matrix.

Returns two objects, a 1-D array containing the eigenvalues of a, and a 2-D square array or matrix (depending on the input type) of the corresponding eigenvectors (in columns).

**Parameters** : a : (... , M, M) array

**Returns** :  $w : (\dots, M)$  ndarray The eigenvalues in ascending order, each repeated according to its multiplicity.

Raises: LinAlgError If the eigenvalue computation does not converge.

## 2.J.2 tests

## 2.K linalg.eigvalsh

### 2.K.1 documentation

Compute the eigenvalues of a Hermitian or real symmetric matrix.

Main difference from eig: the eigenvectors are not computed.

**Parameters** :  $a : (\dots, M, M)$  array\_like

**Returns** :  $w : (\dots, M,)$  ndarray The eigenvalues in ascending order, each repeated according to its multiplicity.

Raises: LinAlgError if the eigenvalue computation does not converge.

### 2.K.2 tests

## 2.L linalg.eigvals

### 2.L.1 documentation

Compute the eigenvalues of a general matrix.

Main difference between eigvals and eig: the eigenvectors are not returned.

**Parameters** :  $a : (\dots, M, M)$  array\_like A complex- or real-valued matrix whose eigenvalues will be computed.

**Returns** :  $w : (\dots, M,)$  ndarray The eigenvalues, each repeated according to its multiplicity. They are not necessarily ordered, nor are they necessarily real for real matrices.

Raises: LinAlgError If the eigenvalue computation does not converge.

### 2.L.2 tests

## 2.M linalg.norm

### 2.M.1 documentation

Matrix or vector norm.

This function is able to return one of eight different matrix norms, or one of an infinite number of vector norms (described below), depending on the value of the ord parameter.

**Parameters** : `x` : array\_like Input array. If axis is None, x must be 1-D or 2-D. `ord` : non-zero int, inf, -inf, fro, nuc, optional Order of the norm (see table under Notes). inf means numpys inf object. `axis` : int, 2-tuple of ints, None, optional If axis is an integer, it specifies the axis of x along which to compute the vector norms. If axis is a 2-tuple, it specifies the axes that hold 2-D matrices, and the matrix norms of these matrices are computed. If axis is None then either a vector norm (when x is 1-D) or a matrix norm (when x is 2-D) is returned. `keepdims` : bool, optional If this is set to True, the axes which are normed over are left in the result as dimensions with size one. With this option the result will broadcast correctly against the original x. New in version 1.10.0.

**Returns** : `n` : float or ndarray Norm of the matrix or vector(s).

## 2.M.2 tests

## 2.N linalg.cond

### 2.N.1 documentation

### 2.N.2 tests

## 2.O linalg.matrix\_rank

### 2.O.1 documentation

Return matrix rank of array using SVD method Rank of the array is the number of singular values of the array that are greater than ‘tol’.

**Parameters** : `M` : (M,), (... , M, N) array\_like input vector or stack of matrices `tol` : (...) array\_like, float, optional threshold below which SVD values are considered zero. If ‘tol’ is None, and “S” is an array with singular values for ‘M’, and “eps” is the epsilon value for datatype of “S”, then ‘tol’ is set to “S.max() \* max(M.shape) \* eps” Broadcasted against the stack of matrices `hermitian` : bool, optional If True, ‘M’ is assumed to be Hermitian (symmetric if real-valued), enabling a more efficient method for finding singular values. Defaults to False.

**Returns** :

### 2.O.2 tests

`test_rank`:

## 2.P linalg.det, linalg.slogdet

### 2.P.1 documentation

Determinants are used to define the characteristic polynomial of a matrix and whether it has a unique solution or not. This function computes the sign and (natural) logarithm of the determinant of an array. A number representing the sign of the determinant. For a real matrix, this is 1, 0, or -1. For a complex matrix, this is a complex number with absolute value 1 (i.e., it is on the unit circle), or else 0. The determinant is computed via LU factorization using the LAPACK routine `z/dgetrf`. The determinant of a 2-D array “[a, b], [c, d]” is “ad - bc”. (`sign, logdet`) = `np.linalg.slogdet(a)`

**Parameters** : An array or matrix with single, double, complex single or complex double type.

**Returns** : A scalar.

## 2.P.2 tests

`test_det`: This tests that the determinant calculation works according to the above.

`test_size_zero`: This tests that the sign of the determinant an empty matrix is a complex number and that the determinant itself is 1.

`test_types`: This tests that the output type of the determinant is the same as the input type, i.e. single, double, csingle and cdouble.

## 2.Q linalg.multidot (Black box tests)

### 2.Q.1 documentation

Compute the dot product of two or more arrays in a single function call, while automatically selecting the fastest evaluation order. ‘multi\_dot’ chains ‘numpy.dot’ and uses optimal parenthesization of the matrices. Depending on the shapes of the matrices, this can speed up the multiplication a lot. If the first argument is 1-D it is treated as a row vector. If the last argument is 1-D it is treated as a column vector. The other arguments must be 2-D.

`TestCases`: Test cases are created so that vectors when multiplied share the same dimensions. When matrices are multiplied they need to be organized so that the first dimension of the first matrix is the same as the second dimension of the second matrix etc.

**Parameters** : Vectors or matrices. They must be organized so that the first dimension of the first matrix is the same as the second dimension of the second matrix etc.

**Returns** : A vector or matrix whose dimension depends on the inputs.

### 2.Q.2 Tests

`test_three_inputs_vectors`: This tests the multidot function with three vectors. The assert is the following: `assert_almost_equal(multi_dot([A, B]), A.dot(B))`

`test_three_inputs_matrices`: This tests the multidot function with three matrices

`test_four_inputs_matrices`: This tests the multidot function with four matrices

`test_shape_vector_first`: This tests the multidot function with a vector with n rows as the first argument followed by three matrices with dimensions n, m and m, n. The shape result sought is the same as the vector, i.e. 1 dimensional with n rows.

`test_shape_vector_last`: This tests the multidot function with a n rows vector as the last argument preceded by three matrices with dimensions m, n and n, m. The shape result sought is m.



`test_shape_vector_first_and_last`: This tests the `multidot` function with `n` rows vector as the first and last arguments with two matrices with dimensions `n, m` and `m, n` in the middle. The shape result sought is `()` since the result is a scalar. `assert_equal(multi_dot([A1d, B, C, D1d]).shape, ())`

`test_types`: This runs the `test_three_inputs_matrices` above using integers, doubles, complex numbers.

### **3 White-Box Test**