Software Testing of Numpy Linear Algebra Library by Team 7

Regina Suraj Johan

January 18, 2018

Contents

1 Introduction

In this project we develop black and white box tests for Python numpys linear algebra package linalg.

The project is relevant since numpy linalg is widely used and linear algebra generally can be regarded as an essential scientific field field. Since linear equations are easy to solve many scientific areas include models where equations are approximated using linear equations. Since solutions to equations in many cases are relevant for practical problem solving linear algebra can be very useful, even in its easiest forms. Some areas in which it is used are module theory, representation theory, ring theory, group theory and Galois theory. In functional theory linear algebra is used to study infinite-dimensional problems. In this field many of the analytical solutions break down even though the linear algebra intuition remains. Linear algebra can be used to understand those areas better.

The following tools are available through the linalg module in numpy:

Core Linear Algebra Tools
———————-
Linear algebra basics:
- norm Vector or matrix norm
- inv Inverse of a square matrix
- solve Solve a linear system of equations
- det Determinant of a square matrix
- lstsq Solve linear least-squares problem
- pinv Pseudo-inverse (Moore-Penrose) calculated using a singular value decomposition
- power Integer power of a square matrix

Eigenvalues and decompositions:
- eig Eigenvalues and vectors of a square matrix
- eigh Eigenvalues and eigenvectors of a Hermitian matrix
- eigvals Eigenvalues of a square matrix
- eigvalsh Eigenvalues of a Hermitian matrix
- qr QR decomposition of a matrix
- svd Singular value decomposition of a matrix
- cholesky Cholesky decomposition of a matrix

Tensor operations:
- tensorsolve Solve a linear tensor equation
- tensorinv Calculate an inverse of a tensor

Exceptions:
- LinAlgError Indicates a failed linear algebra operation

The following linalg functions which we wrote black box tests for:
linalg.dot
linalg. vdot
linalg.inner

linalg.outer
linalg.matmul
linalg.matrix_power
linalg.norm
linalg.matrix_rank
linalg.det, slogdet
linalg.multidot

We also wrote white box tests for:
linalg.multidot

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:

$$dot(a, b)[i, j, k, m] = sum(a[i, j, :] * b[k, :, m]) \tag{1}$$

Paramaters : **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

In order to test this function, the input space is partitioned into three types of test suites:

- **Corner cases:** For the corner cases we try to check what happens when dot product is presented with empty arrays.
- **Basic properties of the dot-product:** This set of tests ensure that the *dot* function obeys the following mathematical properties.
    1. Commutativity: $a \bullet b = b \bullet a$

2. Linear: $a \bullet (\mathbf{r}b + c) = \mathbf{r}(a \bullet b) + (a \bullet c)$
3. Zero dot-product: $a \bullet 0 = 0$
4. Square of product: $a \bullet a = |a^2|$
5. Perpendicular vectors: $c \bullet d = 0$

- **Raises case:** This test is setup to ensure that a ValueError is raised when multiplying vectors of different dimensionality.

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.

Paramaters : **a** : *array_like* If a is complex the complex conjugate is taken before calculation of the dot product.

b : *array_like* Second argument to the dot product.

Returns : **output** : *ndarray* Dot product of a and b. Can be an int, float, or complex depending on the types of a and b.

2.C.2 Tests

For the *vdot* function the input space is partitioned into tests that are formed based on the following division:

- **Basic functionality checks:** These tests involve some regression tests that ensure that the *vdot* function works as intended.
- **Complex numbers:** Some tests work with complex numbers to check commutative and square functionlity of the *vdot* function.
- **Special cases:** The *vdot* function is checked with floats, empty arrays and negative numbers.

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.

Paramaters : **a**, **b** : *array_like* If a and b are nonscalar, their last dimensions must match.

Returns : **out** : *ndarray out.shape = a.shape[:-1] + b.shape[:-1]*

Raises : **ValueError** If the last dimension of a and b has different size.

2.D.2 Tests

In order to test the *inner* product functionality the test suite is divided based on:

- **Regression Tests:** These tests involve some regression tests that ensure that the *inner* function works as intended.
- **Properties of *inner* product:** The inner dot function according to mathworld [1] should obey the following properties:

  1. $\langle u + v, w \rangle = \langle u, w \rangle + \langle v, w \rangle$
  2. $\langle \alpha\ v, w \rangle = \alpha \langle v, w \rangle$
  3. $\langle v, w \rangle = \langle w, v \rangle$
  4. $\langle v, v \rangle \leq 0$

     where u, v, w are vectors and α is a scalar.

  The first suite of tests check to see how the function behaves when a *zero* case is introduced in the arguments and also if the function works with *float* values. The second suite of tests check whether the *inner* dot product verifies with the properties.

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]*

[1] http://mathworld.wolfram.com/InnerProduct.html

2.E.2 Tests

The *outer* product of two vectors behaves as shown below, given two vectors a = [a0, a1, ..., aM] and b = [b0, b1, ..., bN]. The outer product computes :
[[a0*b0 a0*bN... a0*bN] [a1*b0 ...]
[.   ]
[aM*b0...   aM*bN]]

In order to test the *outer* function tests have been made to check the following:

- **Regression Tests:** This test sets up a regression test that checks that the *outer* product works as intended with a basic example that verifies the behaviour of the function.
- **Corner Test:** In order to check the case where the provided vector consists of zero's.
- **Complex value Test:** Since, the documentation claims to also handle complex valued vectors this case makes an *outer* product of two complex valued vectors.
- **Dimension Test:** This test attempts to check how the *outer* product handles vectors of varying sizes.

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 > 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.

Parameters : **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 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. If scalar value is passed.

2.F.2 Tests

In order to test the *matmul* function tests have been made to check the following:

- **Regression Tests:** This test checks that the function works as intended.
- **Commutative and Distributive Tests:** These tests ensure that the *matmul* function according to mathworld [2] obeys properties of commutativity and distributivity.
- **Identity Test:** An identity matrix is multiplied with another matrix to check if the matrix is preserved on matrix multiplication.
- **Raises Checks:** Two tests are setup to ensure that a ValueError is raised when multiplying vectors of different dimensionality and a matrix and scalar are multiplied.

2.G linalg.matrix_power

2.G.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 ¡ 0, the inverse is computed and then raised to the abs(n).

Parameters : **M** : *ndarray or matrix object* Matrix to be powered. Must be square, i.e. M.shape == (m, m), with m a positive integer.
n : *int* The exponent can be any integer or long integer, positive, negative, or zero.

Returns : **M**n** : *ndarray or matrix object* The return value is the same shape and type as M; if the exponent is positive or zero then the type of the elements is the same as those of M. If the exponent is negative the elements are floating-point.

Raises : **LinAlgError** If the matrix is not numerically invertible.

2.G.2 Tests

In order to test the *matrix power* function tests have been made to check the following:

- **Regression Tests:** This test ensures that the matrix when powered by a *numgeq*1 works as intended.
- **Matrix Identity Tests:** This test ensures that a matrix powered by $num == 0$ produces an identity matrix of the same size.
- **Matrix Negative Power Check:** This test checks the *matrix power* fuction when $num \leq -1$.

[2] http://mathworld.wolfram.com/MatrixMultiplication.html

2.H linalg.norm

2.H.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.

Paramaters : 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.H.2 tests

2.I linalg.matrix_rank

2.I.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‘.

Paramaters : 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.I.2 tests

test_simple_case: a simple 3X3 eye matrix is tested

test_scalar: the rank of a scalar input equates to 1.

test_array: same as above but a scalar inside an array.

test_zero_rank: the rank of a matrix with zeros will equate to 0.

test_1_dimensional_matrix: the rank of a 1 dimensinal matrix should be 1.

These tests were written to test the functionality of matrix rank with various simple integer inputs.

2.J linalg.det, linalg.slogdet

2.J.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)

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

Returns : A scalar.

2.J.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.

These tests were written to test matrix determinants with various simple inputs. The determinant function was also evaluated with double, complex single and complex double datatypes.

2.K linalg.multidot (Black box tests)

2.K.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.

Paramaters : 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.K.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.

These tests were written to test the functionality of multidot with various inputs. All test cases are initialized with random values.

2.L Datatype tests

A separate testclass was created to test linalg functions with various datatypes. The functions were run with values of these datatypes and the output was checked to still be the same datatype. The datatypes used were single, double, complex single and complex double. The functions tested with the datatypes were matrix invariant, eig and eigenvalues for normal and hermitian cases, single value decomposition and determinant.

3 White-Box Test

In this section we aim to use what we can see from the functions themselves to satisfy some coverage criteria. To evaluate coverage we will use the coverage.py package. This can evaluate both statement and branch coverage and enumerate which statements or branches were not executed.

The function we have chosen to white box test is the multi_dot() function and it's subsidiary functions _multi_dot() and multi_dot_three().

Node Coverage For node coverage we have the critereon that our tests cause all statements to be executed. Figure 1shows the control flow graph for the functions under test.

Edge Coverage For edge coverage we have the critereon that our tests cause all branches to execute. In this case we have the set of edges
{(1,2),(1,3),(1,4),(4,5),(4,6),(5,6),(6,7),(6,8),(8,9),(8,10),(8,12),(12,10),(10,16),(9,11),(11,13)
,(13,14),(13,15),(15,16),(16,17),(16,18),(16,19)}. Our test requirements are that every edge is contained in at least one of our test paths. // With the below suite of tests we achieve full node and branch coverage.

3.A Test: Arguments<2

For this case the test test_multi_dot_raises was created. See Appendix for test specifics. This test gives us node coverage for {1,2} and branch coverage for {(1,2)}. It returns a raises value error.

3.B Test: Arguments=2

For this case the test test_multi_two was created. See Appendix for test specifics. This test gives us node coverage for {1,3} and branch coverage for {(1,3)}. It returns the dot product of the two.

3.C Test: Arguments=3,dimension of 1st argument = 1

For this case the test test_multi_ndim_10 was created. See Appendix for test specifics. This test gives us node coverage for {1,4,5,6,8,9,11,13,14,16,17} and branch coverage for {(1,4),
(4,5),(6,8),(8,9),(9,11),(11,13),(13,14),(14,16),(16,18)}.

3.D Test: Arguments=3,dimension of last argument = 1

For this case the test test_multi_ndim_01 was created. See Appendix for test specifics.

This test gives us node coverage for {1,4,6,7,8,9,11,13,15,16,18} and branch coverage for {(1,4),

(4,6),(6,7),(7,8),(8,9),(9,11),(11,13),(13,14),(14,16),(16,18)}.

3.E Test: Arguments<3,dimension of first and last argument = 1

For this case the test test_multi_ndim_11 was created. See Appendix for test specifics.

This test gives us node coverage for {1,4,5,6,7,8,10,12,10,16,17} and fulfils the requirements that the edges (8,10),(10,12),(12,10),(16,19) are executed.

3.F Test: Arguments=3,dimension of first and last argument > 1

For this case the test test_multi_ndim_00 was created. See Appendix for test specifics. This test gives us node coverage for {1,4,6,8,9,11,13,14,16,19} and fulfils the final requirement that the edge (16,19) is executed.

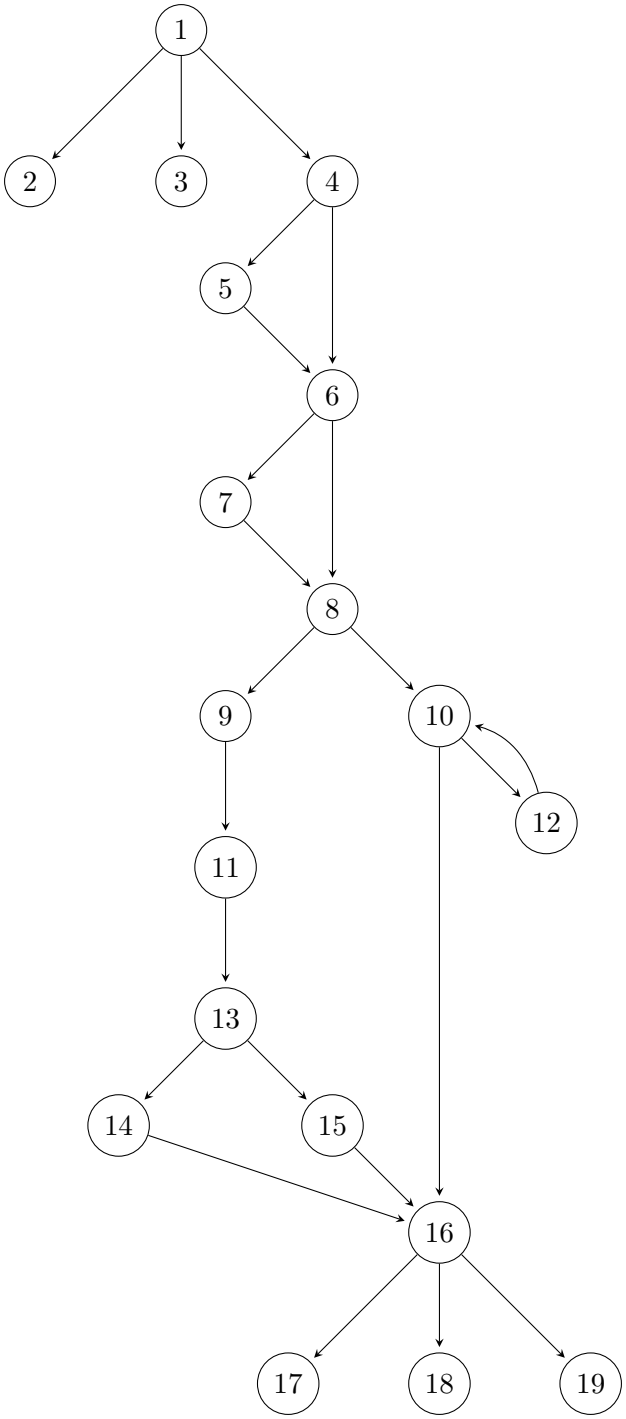

Figure 1: The control flow graph for multi_dot(),multi_dot_three() and _multi_dot()

4 Conclusion

All the tests developed in this project for certain functions of the numpy linalg package yielded good test results and we therefore conclude that they function properly. All the tests do not cover all the possible input parameter datatypes such as complex numbers, which weakens the suite. Developing such tests in a fully comprehensive manner was found to be difficult due to

the various functionalities of the package. Software testing is fun and exciting!

5 Appendix

5.A whitebox

```
import numpy as np
import unittest
import random as rand

from numpy import array, single, double, csingle, cdouble, dot, identity,empty
from numpy import multiply, atleast_2d, inf, asarray, matrix
import linpy



class WhiteBox(unittest.TestCase):
    def setUp(self):
        self.array_2_2_identity = np.identity(2)
        self.array_1 = np.array([[1,2],[3,4]])
        self.array_2 = np.array([[5,6],[7,8]])
        self.array_3 = np.array([[9,10,11],[12,13,14],[15,16,17]])
        self.array_4 = np.array([[6,5],[5,3],[12,15]])
        #self.array_multi_5 = np.rand.rand(3,2)


    def test_multi_dot_raises(self):
        with self.assertRaises(ValueError):
            actual = linpy.multi_dot([self.array_1])

#         try:
#             self.assertFalse(True)
#         except ValueError:
#             pass

    #test 2
    def test_multi_two(self):
        actual = linpy.multi_dot([self.array_1, self.array_2])
        expected1 = np.array([[19, 22], [43, 50]])
        self.assertTrue((actual == expected1).all())

    def test_multi_ndim_01(self):
        actual = linpy.multi_dot([self.sint,self.array_1, self.array_2])
        expected = np.array([434,504])
        self.assertTrue((actual == expected).all())

    def test_multi_ndim_00(self):
        actual = linpy.multi_dot([self.array_4,self.array_1, self.array_2])
        expected = np.array([[329,382],[224,260],[873,1014]])
```

```
        self.assertTrue((actual == expected).all())

    def test_multi_ndim_end1(self):
        actual = linpy.multi_dot([self.array_1, self.array_2,self.sint])
        expected = np.array([287,651])
        self.assertTrue((actual == expected).all())


    def test_multi_ndim_11(self):
        actual = linpy.multi_dot([self.sint,self.array_1,
            self.array_2,self.sfloat])
        expected = np.array([6190.8])
        self.assertAlmostEqual(actual, expected)


if __name__ == '__main__':
    unittest.main()
```

5.B linalg.dot

```
import numpy as np
import unittest

class TestLinAlg(unittest.TestCase):
    def setUp(self):
        """ Basic Identity test """
        self.array_1 = [[1, 0], [0, 1]]
        self.array_2 = [[4, 1], [2, 2]]

        """ Zero Test [1]
(http://gettingsharper.de/2011/11/30/vector-fun-dot-product/)"""
        self.array_zero = [0, 0]

        """ Commutative Test [1]"""
        self.array_com_1 = [-3.22 , 2.25, -0.13]
        self.array_com_2 = [0.0 , -6.7, 10.0]

        """Linear Test [1]"""
        self.array_com_3 = [12.4, -1.7, 3.15]
        self.scalar = 0.22

        """ Square Test [1]"""

        """ Perpendicular Test [1]"""
        self.array_per_1 = [2.0 , 1.0, 4.0]
        self.array_per_2 = [1.0 , -1.0, -0.25]

        """
            Testing numpy.linalg.dot() function
        """
```

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

    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_perpendicuar(self):
        actual = np.dot(self.array_per_1, self.array_per_2)
        expected = 0
        self.assertTrue((actual == expected).all())

if __name__ == '__main__':
    unittest.main()
```

5.C linalg.vdot

```
import numpy as np
import unittest

class TestLinAlg(unittest.TestCase):
    def setUp(self):
```

```
        self.array_a = np.array([[1, 4], [5, 6]])
        self.array_b = np.array([[4, 1], [2, 2]])

        self.array_a_float = np.array([1.0, 4.5])
        self.array_b_float = np.array([3.0, 2.5])


        """
            Testing numpy.linalg.vdot() function
        """
    def setupForComplex(self):
        self.complex_a = np.array([1+2j,3+4j])
        self.complex_b = np.array([5+6j,7+8j])
        self.complex_c = np.array([-5-6j,-7-8j])

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

        self.assertTrue(actual == expected)

    def test_vdot_complexSquare(self):
        self.setupForComplex()

        actual = np.vdot(self.complex_a, self.complex_a)
        expected = 30+0j

        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)




if __name__ == '__main__':
    unittest.main()
```

5.D linalg.inner

```
import numpy as np
import unittest

class TestLinAlg(unittest.TestCase):
    def setUp(self):

        self.array_a = np.array([1, 2, 3])
        self.array_b = np.array([0, 1, 0])

        self.array_a_float = np.array([1.0, 2.0, 4.5])
        self.array_b_float = np.array([3.0, 3.5, 2.5])

        """Inner Product Wolfram"""
        self.vector_u = np.array([1,2,3])
        self.vector_v = np.array([1,2,1])
        self.vector_w = np.array([4,5,6])
        self.scalar = 5
        self.vector_zero = np.array([0, 0, 0])

        """
            Testing numpy.linalg.inner() function
        """

    def test_inner_simple(self):
        actual = np.inner(self.array_a, self.array_b)
        expected = 2
        self.assertTrue(actual == expected)
```

```
    def test_inner_zero(self):
        actual = np.inner(self.array_a, [0, 0, 0])
        expected = 0
        self.assertTrue(actual == expected)

    def test_inner_float(self):
        actual = np.inner(self.array_a_float, self.array_b_float)
        expected = 21.25
        self.assertTrue(actual == expected)

    """http://mathworld.wolfram.com/InnerProduct.html"""
    def test_inner_prop1(self):
        actual = np.inner(np.add(self.vector_u,self.vector_v), self.vector_w)
        expected = np.add(np.inner(self.vector_u,self.vector_w),
                          np.inner(self.vector_v,self.vector_w))
        self.assertTrue(actual == expected)

    def test_inner_prop2(self):
        actual = np.inner(self.scalar * self.vector_v, self.vector_w)
        expected = self.scalar * np.inner(self.vector_v, self.vector_w)
        self.assertTrue(actual == expected)

    def test_inner_prop3(self):
        actual = np.inner(self.vector_v, self.vector_w)
        expected = np.inner(self.vector_w, self.vector_v)
        self.assertTrue(actual == expected)

    def test_inner_prop4(self):
        actual = np.inner(self.vector_zero, self.vector_zero)
        expected = 0
        self.assertTrue(actual == expected)

    def test_inner_raises(self):
        with self.assertRaises(ValueError):
            actual = np.inner([2, 2, 3], [2, 1])


if __name__ == '__main__':
    unittest.main()
```

5.E linalg.outer

```
import numpy as np
import unittest

class TestLinAlg(unittest.TestCase):
    def setUp(self):

        self.vector_a = np.array(['a', 'b', 'c'], dtype=object)
```

```
        self.vector_b = np.array([1, 2, 3])
        self.vector_c = np.array([1, 2])
        self.vector_zero = np.array([0, 0, 0], dtype=object)

        self.vector_com_a = np.array([1+2j, 2+3j])
        self.vector_com_b = np.array([1+2j, 2+3j])

        """
            Testing numpy.linalg.outer() function
        """

    def test_outer_simple(self):
        actual = np.outer(self.vector_a, self.vector_b)
        expected = [['a', 'aa', 'aaa'],['b', 'bb', 'bbb'],['c', 'cc', 'ccc']]
        self.assertTrue((actual == expected).all())

    def test_outer_zero(self):
        actual = np.outer(self.vector_b, self.vector_zero)
        expected = np.zeros((3, 3))
        self.assertTrue((actual == expected).all())

    def test_outer_complex(self):
        actual = np.outer(self.vector_com_a, self.vector_com_b)
        expected = np.array([[-3. +4.j, -4. +7.j],[-4. +7.j, -5.+12.j]])
        self.assertTrue((actual == expected).all())

    def test_outer_dimensions(self):
        actual = np.outer(self.vector_a, self.vector_c)
        expected = [['a', 'aa'],['b', 'bb'],['c', 'cc']]
        self.assertTrue((actual == expected).all())

if __name__ == '__main__':
    unittest.main()
```

5.F linalg.matmul

```
import numpy as np
import unittest

class TestLinAlg(unittest.TestCase):
    def setUp(self):

        self.vector_a = np.array([[1, 0],[0, 1]])
        self.vector_b = np.array([[4, 1],[2, 2]])

        self.vector_2d = np.array([[1, 0],[0, 1]])
        self.vector_1d = np.array([1,2])

        """http://mathworld.wolfram.com/MatrixMultiplication.html"""
        self.matrix_a = np.array([[1, 0],[0, 1]])
```

```
        self.matrix_b = np.array([[4, 1],[2, 2]])
        self.matrix_c = np.array([[3, 2],[2, 3]])

        self.matrix_diag_a = np.array([[1,0,0],[0,1,0],[0,0,1]])
        self.matrix_diag_b = np.array([[2,0,0],[0,2,0],[0,0,2]])

        """
            Testing numpy.linalg.matmul() function
        """

    def test_matmul_simple(self):
        actual = np.matmul(self.vector_a, self.vector_b)
        expected = np.array([[4,1],[2,2]])
        self.assertTrue((actual == expected).all())

    def test_matmul_mix(self):
        actual = np.matmul(self.vector_2d, self.vector_1d)
        expected = np.array([1,2])
        self.assertTrue((actual == expected).all())

    def test_matmul_distributivity(self):
        actual = np.matmul(self.matrix_a, np.add(self.matrix_b, self.matrix_c))
        expected = np.add(np.matmul(self.matrix_a, self.matrix_b),
                          np.matmul(self.matrix_a, self.matrix_c))
        self.assertTrue((actual == expected).all())

    def test_matmul_diag_commutative(self):
        actual = np.matmul(self.matrix_diag_a, self.matrix_diag_b)
        expected = np.matmul(self.matrix_diag_b, self.matrix_diag_a)
        self.assertTrue((actual == expected).all())

    def test_matmul_raises(self):
        with self.assertRaises(ValueError):
            actual = np.matmul([2, 2, 3], [2, 1])

    def test_matmul_scalar_raises(self):
        with self.assertRaises(ValueError):
            actual = np.matmul([2, 2, 3], 2)

if __name__ == '__main__':
    unittest.main()
```

5.G linalg.matrixpower

```
import numpy as np
from numpy import linalg as LA
import unittest

class TestLinAlg(unittest.TestCase):
    def setUp(self):
```

```
        self.matrix_a = np.array([[4, 1, 2],[2, 2, 1],[3, 2, 2]])
        self.matrix_b = np.matrix([[0, 1],[-1, 0]])

        """
            Testing numpy.linalg.matrix_power() function
        """

    def test_matrixpower_simple(self):
        actual = LA.matrix_power(self.matrix_a, 3)
        expected = np.array([[155, 70, 84],[100, 47, 54],[146, 68, 79]])
        self.assertTrue((actual == expected).all())

    def test_matrixpower_negative(self):
        actual = LA.matrix_power(self.matrix_b, -3)
        expected = np.array([[0., 1.],[-1., 0.]])
        self.assertTrue((actual == expected).all())

    def test_matrixpower_identity(self):
        actual = LA.matrix_power(self.matrix_a, 0)
        expected = np.array([[1, 0, 0],[0, 1, 0],[0, 0, 1]])
        self.assertTrue((actual == expected).all())

if __name__ == '__main__':
    unittest.main()
```

5.H test_multi_dot

```
CASES = []

dimension1 = np.random.randint(1, 12)
dimension2 = np.random.randint(1, 12)

CASES += [np.random.random((dimension1, dimension2)), # 0
                np.random.random((dimension2, dimension1)),
                np.random.random((dimension1, dimension2)),
                np.random.random((dimension2, dimension1)),
                np.random.random(dimension1),
                np.random.random(dimension1), # 5


class TestMultiDot(unittest.TestCase):

    def test_two_inputs_vectors(self):
        A = CASES[4]
        B = CASES[5]

        assert_almost_equal(multi_dot([A, B]), A.dot(B))
        assert_almost_equal(multi_dot([A, B]), np.dot(A, B))


    def test_three_inputs_vectors(self):

        A = CASES[4]
        B = CASES[5]
        C = CASES[2]

        # running multi-dot on three equal size vectors should result in a multi_dot error
        try:
            assert_almost_equal(multi_dot([A, B, C]), np.dot(np.dot(A, B), C))
        except Exception:
            pass

 def test_three_inputs_matrices(self):
     A = CASES[0]
     B = CASES[1]
     C = CASES[2]

     assert_almost_equal(multi_dot([A, B, C]), A.dot(B).dot(C))
     assert_almost_equal(multi_dot([A, B, C]), np.dot(A, np.dot(B, C)))

     #when the inputs are in the wrong order
     try:
         assert_almost_equal(multi_dot([A, C, B]), np.dot(np.dot(A, C), C))
     except Exception:
         pass


 def test_four_inputs_matrices(self):
     A = CASES[0]
     B = CASES[1]
     C = CASES[2]
     D = CASES[3]

     assert_almost_equal(multi_dot([A, B, C, D]), A.dot(B).dot(C).dot(D))

     # when the inputs are in the wrong order
     try:
         assert_almost_equal(multi_dot([A, C, B]), np.dot(np.dot(A, C), C))
     except Exception:
         pass

        def test_vector_as_first_argument(self):
            # The first argument can be 1-D
            A1d = np.random.random(2)  # 1-D
            B = np.random.random((2, 6))
            C = np.random.random((6, 2))
            D = np.random.random((2, 2))

            # the result should be 1-D
            assert_equal(multi_dot([A1d, B, C, D]).shape, (2,))

        def test_vector_as_last_argument(self):
            # The last argument can be 1-D
            A = np.random.random((6, 2))
            B = np.random.random((2, 6))
            C = np.random.random((6, 2))
            D1d = np.random.random(2)  # 1-D

            # the result should be 1-D
            assert_equal(multi_dot([A, B, C, D1d]).shape, (6,))

        def test_vector_as_first_and_last_argument(self):
            # The first and last arguments can be 1-D
            A1d = np.random.random(2)  # 1-D
            B = np.random.random((2, 6))
            C = np.random.random((6, 2))
            D1d = np.random.random(2)  # 1-D

            # the result should be a scalar
            assert_equal(multi_dot([A1d, B, C, D1d]).shape, ())
```

5.I test_matrix_rank

```
class TestMatrixRank(Unittest.TestCase):

    def test_simple_case(self):
        # the rank of a 3X3 matrix is 3
        assert_equal(linalg.matrix_rank(np.eye(3)), 3)

    def test_scalar(self):
        assert_equal(matrix_rank(1), 1)

    def test_array(self):
        # the rank of an array with a single value should be one
        assert_equal(matrix_rank([1]), 1)

    def test_zero_rank(self):
        assert_equal(linalg.matrix_rank(np.zeros((4, 4))), 0)

    def test_1_dimensional_matrix(self):
        assert_equal(matrix_rank([1, 0, 0, 0]), 1)
```

5.J test_matrix_determinant

```
class TestDet(Unittest.TestCase):
    def test_empty_simple(self):
        assert_equal(linalg.det([[0.0]]), 0.0)

    def test_det_empty_types(self):
        assert_equal(type(linalg.det([[0.0]])), double)
        assert_equal(linalg.det([[0.0j]]), 0.0)
        assert_equal(type(linalg.det([[0.0j]])), cdouble)

    def test_slogdet_empty_types(self):
        assert_equal(linalg.slogdet([[0.0j]]), (0.0j, -inf))
        assert_equal(type(linalg.slogdet([[0.0j]])[0]), cdouble)
        assert_equal(type(linalg.slogdet([[0.0j]])[1]), double)
        assert_equal(linalg.slogdet([[0.0]]), (0.0, -inf))
        assert_equal(type(linalg.slogdet([[0.0]])[0]), double)
        assert_equal(type(linalg.slogdet([[0.0]])[1]), double)

    def test_size_simple(self):
        # create an empty array
        empty_array = np.zeros((0, 0))
        # compute the determinant
        determinant_result = linalg.det(empty_array)
        # make sure the result is 1
        assert_equal(determinant_result, 1.)

        #do the same for slogdet
        determinant_result = linalg.slogdet(empty_array)

        #the result in this case should be (1, 0)
        assert_equal(determinant_result, (1, 0))
```

5.K test_datatypes

```
# GLOBAL FUNCTIONS
def get_real_dtype(dtype):
    return {single: single, double: double,
            csingle: single, cdouble: double}[dtype]



class LinalgCase(object):
    def __init__(self, name, a, b, tags=set()):


        assert_(isinstance(name, str))
        self.name = name
        self.a = a
        self.b = b
        self.datatype = frozenset(tags)  # prevent shared tags


CASES = []

CASES += [LinalgCase("single",
                     array([1., 1., 1.], dtype=single),
                     array([1., 1., 1.], dtype=single)),
          LinalgCase("double",
                     array([[1., 2.], [3., 4.]], dtype=double),
                     array([2., 1.], dtype=double)),
          ]

class TestDatatypes(unittest.TestCase):

    def test_invTypes(self):
        x = np.array([[1, 0.5], [0.5, 1]], dtype=single)
        assert_equal(linalg.solve(x, x).dtype, single)

    def test_EigvalTypes(self):
        def check(dtype):
            x = np.array([[1, 0.5], [0.5, 1]], dtype=dtype)
            assert_equal(linalg.eigvals(x).dtype, dtype)
        for dtype in [single, double, csingle, cdouble]:
            yield check, dtype

    def test_EigvalHermitianTypes(self):
        def check(dtype):
            x = np.array([[1, 0.5], [0.5, 1]], dtype=dtype)
            w = np.linalg.eigvalsh(x)
            assert_equal(w.dtype, get_real_dtype(dtype))
        for dtype in [single, double, csingle, cdouble]:
            yield check, dtype

    def test_EigTypes(self):
        def check(dtype):
            x = np.array([[1, 0.5], [0.5, 1]], dtype=dtype)
            w, v = np.linalg.eig(x)
            assert_equal(w.dtype, dtype)
            assert_equal(v.dtype, dtype)

        for dtype in [single, double, csingle, cdouble]:
            yield check, dtype
    def test_SVDtypes(self):
        def check(dtype):
            x = np.array([[1, 0.5], [0.5, 1]], dtype=dtype)
            u, s, vh = linalg.svd(x)
            assert_equal(u.dtype, dtype)
        for dtype in [single, double, csingle, cdouble]:
            yield check, dtype

    def test_determinantTypes(self):
        def check(dtype):
            x = np.array([[1, 0.5], [0.5, 1]], dtype=dtype)
            assert_equal(np.linalg.det(x).dtype, dtype)
        for dtype in [single, double, csingle, cdouble]:
            yield check, dtype

if __name__ == '__main__':
    unittest.main()
```