Linear Algebra (`scipy.linalg`)

When SciPy is built using the optimized ATLAS LAPACK and BLAS libraries, it has very fast linear algebra capabilities. If you dig deep enough, all of the raw LAPACK and BLAS libraries are available for your use for even more speed. In this section, some easier-to-use interfaces to these routines are described.

All of these linear algebra routines expect an object that can be converted into a 2-D array. The output of these routines is also a 2-D array.

scipy.linalg vs numpy.linalg

`scipy.linalg` contains all the functions in numpy.linalg. plus some other more advanced ones not contained in `numpy.linalg`.

Another advantage of using `scipy.linalg` over `numpy.linalg` is that it is always compiled with BLAS/LAPACK support, while for NumPy this is optional. Therefore, the SciPy version might be faster depending on how NumPy was installed.

Therefore, unless you don't want to add `scipy` as a dependency to your `numpy` program, use `scipy.linalg` instead of `numpy.linalg`.

numpy.matrix vs 2-D numpy.ndarray

The classes that represent matrices, and basic operations, such as matrix multiplications and transpose are a part of `numpy`. For convenience, we summarize the differences between `numpy.matrix` and `numpy.ndarray` here.

`numpy.matrix` is matrix class that has a more convenient interface than `numpy.ndarray` for matrix operations. This class supports, for example, MATLAB-like creation syntax via the semicolon, has matrix multiplication as default for the `*` operator, and contains `I` and `T` members that serve as shortcuts for inverse and transpose:

```
>>> import numpy as np
>>> A = np.mat('[1 2;3 4]')
>>> A
matrix([[1, 2],
        [3, 4]])
>>> A.I
matrix([[-2. ,  1. ],
        [ 1.5, -0.5]])
>>> b = np.mat('[5 6]')
>>> b
matrix([[5, 6]])
>>> b.T
matrix([[5],
        [6]])
>>> A*b.T
matrix([[17],
        [39]])
```

Despite its convenience, the use of the `numpy.matrix` class is discouraged, since it adds nothing that cannot be accomplished with 2-D `numpy.ndarray` objects, and may lead to a confusion of which class is being used. For example, the above code can be rewritten as:

```
>>> import numpy as np
>>> from scipy import linalg
>>> A = np.array([[1,2],[3,4]])
>>> A
array([[1, 2],
       [3, 4]])
>>> linalg.inv(A)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
>>> b = np.array([[5,6]]) #2D array
>>> b
array([[5, 6]])
>>> b.T
array([[5],
       [6]])
>>> A*b #not matrix multiplication!
array([[ 5, 12],
       [15, 24]])
>>> A.dot(b.T) #matrix multiplication
array([[17],
       [39]])
>>> b = np.array([5,6]) #1D array
>>> b
array([5, 6])
>>> b.T  #not matrix transpose!
array([5, 6])
>>> A.dot(b)  #does not matter for multiplication
array([17, 39])
```

`scipy.linalg` operations can be applied equally to `numpy.matrix` or to 2D `numpy.ndarray` objects.

Basic routines

Finding the inverse

The inverse of a matrix $\mathbf{A}$ is the matrix $\mathbf{B}$, such that $\mathbf{AB} = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix consisting of ones down the main diagonal. Usually, $\mathbf{B}$ is denoted $\mathbf{B} = \mathbf{A}^{-1}$. In SciPy, the matrix inverse of the NumPy array, A, is obtained using **`linalg.inv`** `(A)`, or using `A.I` if `A` is a Matrix. For example, let

$$\mathbf{A} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 5 & 1 \\ 2 & 3 & 8 \end{bmatrix},$$

then

$$\mathbf{A^{-1}} = \frac{1}{25} \begin{bmatrix} -37 & 9 & 22 \\ 14 & 2 & -9 \\ 4 & -3 & 1 \end{bmatrix} = \begin{bmatrix} -1.48 & 0.36 & 0.88 \\ 0.56 & 0.08 & -0.36 \\ 0.16 & -0.12 & 0.04 \end{bmatrix}.$$

The following example demonstrates this computation in SciPy

```
>>> import numpy as np
>>> from scipy import linalg
>>> A = np.array([[1,3,5],[2,5,1],[2,3,8]])
>>> A
array([[1, 3, 5],
       [2, 5, 1],
       [2, 3, 8]])
>>> linalg.inv(A)
array([[-1.48,  0.36,  0.88],
       [ 0.56,  0.08, -0.36],
       [ 0.16, -0.12,  0.04]])
>>> A.dot(linalg.inv(A)) #double check
array([[  1.00000000e+00,  -1.11022302e-16,  -5.55111512e-17],
       [  3.05311332e-16,   1.00000000e+00,   1.87350135e-16],
       [  2.22044605e-16,  -1.11022302e-16,   1.00000000e+00]])
```

Solving a linear system

Solving linear systems of equations is straightforward using the scipy command **`linalg.solve`**. This command expects an input matrix and a right-hand side vector. The solution vector is then computed. An option for entering a symmetric matrix is offered, which can speed up the processing when applicable. As an example, suppose it is desired to solve the following simultaneous equations:

$$x + 3y + 5z = 10$$
$$2x + 5y + z = 8$$
$$2x + 3y + 8z = 3$$

We could find the solution vector using a matrix inverse:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 5 & 1 \\ 2 & 3 & 8 \end{bmatrix}^{-1} \begin{bmatrix} 10 \\ 8 \\ 3 \end{bmatrix} = \frac{1}{25} \begin{bmatrix} -232 \\ 129 \\ 19 \end{bmatrix} = \begin{bmatrix} -9.28 \\ 5.16 \\ 0.76 \end{bmatrix}.$$

However, it is better to use the linalg.solve command, which can be faster and more numerically stable. In this case, it, however, gives the same answer as shown in the following example:

```
>>> import numpy as np
>>> from scipy import linalg
>>> A = np.array([[1, 2], [3, 4]])
>>> A
array([[1, 2],
       [3, 4]])
>>> b = np.array([[5], [6]])
>>> b
array([[5],
       [6]])
>>> linalg.inv(A).dot(b)  # slow
array([[-4. ],
       [ 4.5]])
>>> A.dot(linalg.inv(A).dot(b)) - b  # check
array([[  8.88178420e-16],
       [  2.66453526e-15]])
>>> np.linalg.solve(A, b)  # fast
array([[-4. ],
       [ 4.5]])
>>> A.dot(np.linalg.solve(A, b)) - b  # check
array([[ 0.],
       [ 0.]])
```

Finding the determinant

The determinant of a square matrix $\mathbf{A}$ is often denoted $|\mathbf{A}|$ and is a quantity often used in linear algebra. Suppose a_{ij} are the elements of the matrix $\mathbf{A}$ and let $M_{ij} = |\mathbf{A}_{ij}|$ be the determinant of the matrix left by removing the i^{th} row and j^{th} column from $\mathbf{A}$. Then, for any row i,

$$|\mathbf{A}| = \sum_j (-1)^{i+j} a_{ij} M_{ij}.$$

This is a recursive way to define the determinant, where the base case is defined by accepting that the determinant of a 1×1 matrix is the only matrix element. In SciPy the determinant can be calculated with `linalg.det`. For example, the determinant of

$$\mathbf{A} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 5 & 1 \\ 2 & 3 & 8 \end{bmatrix}$$

is

$$\begin{aligned} |\mathbf{A}| &= 1 \begin{vmatrix} 5 & 1 \\ 3 & 8 \end{vmatrix} - 3 \begin{vmatrix} 2 & 1 \\ 2 & 8 \end{vmatrix} + 5 \begin{vmatrix} 2 & 5 \\ 2 & 3 \end{vmatrix} \\ &= 1\,(5 \cdot 8 - 3 \cdot 1) - 3\,(2 \cdot 8 - 2 \cdot 1) + 5\,(2 \cdot 3 - 2 \cdot 5) = -25. \end{aligned}$$

.

In SciPy, this is computed as shown in this example:

```
>>> import numpy as np
>>> from scipy import linalg
>>> A = np.array([[1,2],[3,4]])
>>> A
array([[1, 2],
       [3, 4]])
>>> linalg.det(A)
-2.0
```

Computing norms

Matrix and vector norms can also be computed with SciPy. A wide range of norm definitions are available using different parameters to the order argument of `linalg.norm`. This function takes a rank-1 (vectors) or a rank-2 (matrices) array and an optional order argument (default is 2). Based on these inputs, a vector or matrix norm of the requested order is computed.

For vector *x*, the order parameter can be any real number including `inf` or `-inf`. The computed norm is

$$\|\mathbf{x}\| = \begin{cases} \max |x_i| & \mathrm{ord} = \inf \\ \min |x_i| & \mathrm{ord} = -\inf \\ \left(\sum_i |x_i|^{\mathrm{ord}}\right)^{1/\mathrm{ord}} & |\mathrm{ord}| < \infty. \end{cases}$$

For matrix $\mathbf{A}$, the only valid values for norm are $\pm 2, \pm 1,$ $\pm$ inf, and 'fro' (or 'f') Thus,

$$\|\mathbf{A}\| = \begin{cases} \max_i \sum_j |a_{ij}| & \mathrm{ord} = \inf \\ \min_i \sum_j |a_{ij}| & \mathrm{ord} = -\inf \\ \max_j \sum_i |a_{ij}| & \mathrm{ord} = 1 \\ \min_j \sum_i |a_{ij}| & \mathrm{ord} = -1 \\ \max \sigma_i & \mathrm{ord} = 2 \\ \min \sigma_i & \mathrm{ord} = -2 \\ \sqrt{\mathrm{trace}\left(\mathbf{A}^H \mathbf{A}\right)} & \mathrm{ord} = \mathrm{'fro'} \end{cases}$$

where $\boldsymbol{\sigma_i}$ are the singular values of $\mathbf{A}$.

Examples:

```
>>> import numpy as np
>>> from scipy import linalg
>>> A=np.array([[1,2],[3,4]])
>>> A
array([[1, 2],
       [3, 4]])
>>> linalg.norm(A)
5.4772255750516612
>>> linalg.norm(A,'fro') # frobenius norm is the default
5.4772255750516612
>>> linalg.norm(A,1) # L1 norm (max column sum)
6
>>> linalg.norm(A,-1)
4
>>> linalg.norm(A,np.inf) # L inf norm (max row sum)
7
```

Solving linear least-squares problems and pseudo-inverses

Linear least-squares problems occur in many branches of applied mathematics. In this problem, a set of linear scaling coefficients is sought that allows a model to fit the data. In particular, it is assumed that data $\boldsymbol{y_i}$ is related to data $\mathbf{x_i}$ through a set of coefficients $\boldsymbol{c_j}$ and model functions $\boldsymbol{f_j}\left(\mathbf{x_i}\right)$ via the model

$$\boldsymbol{y_i} = \sum_j \boldsymbol{c_j} \boldsymbol{f_j}\left(\mathbf{x}_i\right) + \boldsymbol{\epsilon_i},$$

where $\boldsymbol{\epsilon_i}$ represents uncertainty in the data. The strategy of least squares is to pick the coefficients $\boldsymbol{c_j}$ to minimize

$$J\left(\mathbf{c}\right) = \sum_i \left| y_i - \sum_j c_j f_j\left(x_i\right) \right|^2.$$

Theoretically, a global minimum will occur when

$$\frac{\partial J}{\partial c_n^*} = 0 = \sum_i \left(y_i - \sum_j c_j f_j\left(x_i\right) \right) \left(-f_n^*\left(x_i\right)\right)$$

or

$$\sum_j c_j \sum_i f_j\left(x_i\right) f_n^*\left(x_i\right) = \sum_i y_i f_n^*\left(x_i\right)$$

$$\mathbf{A}^H \mathbf{A} \mathbf{c} = \mathbf{A}^H \mathbf{y}$$

,
where

$$\{\mathbf{A}\}_{ij} = f_j(x_i).$$

When $\mathbf{A^H A}$ is invertible, then

$$\mathbf{c} = \left(\mathbf{A}^H\mathbf{A}\right)^{-1}\mathbf{A}^H\mathbf{y} = \mathbf{A}^\dagger\mathbf{y},$$

where $\mathbf{A}^\dagger$ is called the pseudo-inverse of $\mathbf{A}$. Notice that using this definition of $\mathbf{A}$ the model can be written

$$\mathbf{y} = \mathbf{A}\mathbf{c} + \boldsymbol{\epsilon}.$$

The command `linalg.lstsq` will solve the linear least-squares problem for $\mathbf{c}$ given $\mathbf{A}$ and $\mathbf{y}$. In addition, `linalg.pinv` will find $\mathbf{A}^\dagger$ given $\mathbf{A}$.

The following example and figure demonstrate the use of `linalg.lstsq` and `linalg.pinv` for solving a data-fitting problem. The data shown below were generated using the model:

$$y_i = c_1 e^{-x_i} + c_2 x_i,$$

where $x_i = 0.1i$ for $i = 1 \ldots 10$, $c_1 = 5$, and $c_2 = 4$. Noise is added to y_i and the coefficients c_1 and c_2 are estimated using linear least squares.

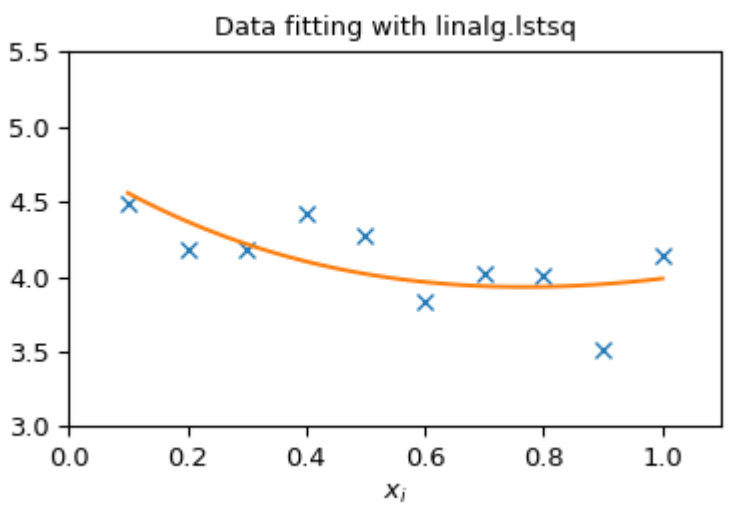
```
>>> import numpy as np
>>> from scipy import linalg
>>> import matplotlib.pyplot as plt
>>> rng = np.random.default_rng()
```

```
>>> c1, c2 = 5.0, 2.0
>>> i = np.r_[1:11]
>>> xi = 0.1*i
>>> yi = c1*np.exp(-xi) + c2*xi
>>> zi = yi + 0.05 * np.max(yi) * rng.standard_normal(len(yi))
```

```
>>> A = np.c_[np.exp(-xi)[:, np.newaxis], xi[:, np.newaxis]]
>>> c, resid, rank, sigma = linalg.lstsq(A, zi)
```

```
>>> xi2 = np.r_[0.1:1.0:100j]
>>> yi2 = c[0]*np.exp(-xi2) + c[1]*xi2
```

```
>>> plt.plot(xi,zi,'x',xi2,yi2)
>>> plt.axis([0,1.1,3.0,5.5])
>>> plt.xlabel('$x_i$')
>>> plt.title('Data fitting with linalg.lstsq')
>>> plt.show()
```



Generalized inverse

The generalized inverse is calculated using the command `linalg.pinv`. Let $\mathbf{A}$ be an $M \times N$ matrix, then if $M > N$, the generalized inverse is

$$\mathbf{A}^\dagger = \left(\mathbf{A}^H\mathbf{A}\right)^{-1}\mathbf{A}^H,$$

while if $M < N$ matrix, the generalized inverse is

$$\mathbf{A}^{\#} = \mathbf{A}^{H}\left(\mathbf{A}\mathbf{A}^{H}\right)^{-1}.$$

In the case that $M = N$, then

$$\mathbf{A}^{\dagger} = \mathbf{A}^{\#} = \mathbf{A}^{-1},$$

as long as $\mathbf{A}$ is invertible.

Decompositions

In many applications, it is useful to decompose a matrix using other representations. There are several decompositions supported by SciPy.

Eigenvalues and eigenvectors

The eigenvalue-eigenvector problem is one of the most commonly employed linear algebra operations. In one popular form, the eigenvalue-eigenvector problem is to find for some square matrix $\mathbf{A}$ scalars $\boldsymbol{\lambda}$ and corresponding vectors $\mathbf{v}$, such that

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}.$$

For an $N \times N$ matrix, there are N (not necessarily distinct) eigenvalues — roots of the (characteristic) polynomial

$$|\mathbf{A} - \lambda\mathbf{I}| = 0.$$

The eigenvectors, $\mathbf{v}$, are also sometimes called right eigenvectors to distinguish them from another set of left eigenvectors that satisfy

$$\mathbf{v}_L^H\mathbf{A} = \lambda\mathbf{v}_L^H$$

or

$$\mathbf{A}^H\mathbf{v}_L = \lambda^*\mathbf{v}_L.$$

With its default optional arguments, the command `linalg.eig` returns $\boldsymbol{\lambda}$ and $\mathbf{v}$. However, it can also return $\mathbf{v}_L$ and just $\boldsymbol{\lambda}$ by itself (`linalg.eigvals` returns just $\boldsymbol{\lambda}$ as well).

In addition, `linalg.eig` can also solve the more general eigenvalue problem

$$\begin{aligned}\mathbf{A}\mathbf{v} &= \lambda\mathbf{B}\mathbf{v}\\ \mathbf{A}^H\mathbf{v}_L &= \lambda^*\mathbf{B}^H\mathbf{v}_L\end{aligned}$$

for square matrices $\mathbf{A}$ and $\mathbf{B}$. The standard eigenvalue problem is an example of the general eigenvalue problem for $\mathbf{B} = \mathbf{I}$. When a generalized eigenvalue problem can be solved, it provides a decomposition of $\mathbf{A}$ as

$$\mathbf{A} = \mathbf{B}\mathbf{V}\boldsymbol{\Lambda}\mathbf{V}^{-1},$$

where $\mathbf{V}$ is the collection of eigenvectors into columns and $\boldsymbol{\Lambda}$ is a diagonal matrix of eigenvalues.

By definition, eigenvectors are only defined up to a constant scale factor. In SciPy, the scaling factor for the eigenvectors is chosen so that $\|\mathbf{v}\|^2 = \sum_i v_i^2 = 1$.

As an example, consider finding the eigenvalues and eigenvectors of the matrix

$$\mathbf{A} = \begin{bmatrix}1 & 5 & 2\\ 2 & 4 & 1\\ 3 & 6 & 2\end{bmatrix}.$$

The characteristic polynomial is

$$\begin{aligned}|\mathbf{A} - \lambda\mathbf{I}| &= (1-\lambda)\left[(4-\lambda)(2-\lambda) - 6\right] -\\ &\quad 5\left[2(2-\lambda) - 3\right] + 2\left[12 - 3(4-\lambda)\right]\\ &= -\lambda^3 + 7\lambda^2 + 8\lambda - 3.\end{aligned}$$

The roots of this polynomial are the eigenvalues of $\mathbf{A}$:

$$\lambda_1 = 7.9579$$
$$\lambda_2 = -1.2577$$
$$\lambda_3 = 0.2997.$$

The eigenvectors corresponding to each eigenvalue can be found using the original equation. The eigenvectors associated with these eigenvalues can then be found.

```
>>> import numpy as np
>>> from scipy import linalg
>>> A = np.array([[1, 2], [3, 4]])
>>> la, v = linalg.eig(A)
>>> l1, l2 = la
>>> print(l1, l2)   # eigenvalues
(-0.372281323269014+0j) (5.372281323269014+0j)
>>> print(v[:, 0])   # first eigenvector
[-0.82456484  0.56576746]
>>> print(v[:, 1])   # second eigenvector
[-0.41597356 -0.90937671]
>>> print(np.sum(abs(v**2), axis=0))  # eigenvectors are unitary
[1. 1.]
>>> v1 = np.array(v[:, 0]).T
>>> print(linalg.norm(A.dot(v1) - l1*v1))  # check the computation
3.23682852457e-16
```

Singular value decomposition

Singular value decomposition (SVD) can be thought of as an extension of the eigenvalue problem to matrices that are not square. Let $\mathbf{A}$ be an $M \times N$ matrix with M and N arbitrary. The matrices $\mathbf{A}^H\mathbf{A}$ and $\mathbf{A}\mathbf{A}^H$ are square hermitian matrices [1] of size $N \times N$ and $M \times M$, respectively. It is known that the eigenvalues of square hermitian matrices are real and non-negative. In addition, there are at most $\mathbf{min}\,(M, N)$ identical non-zero eigenvalues of $\mathbf{A}^H\mathbf{A}$ and $\mathbf{A}\mathbf{A}^H$. Define these positive eigenvalues as σ_i^2. The square-root of these are called singular values of $\mathbf{A}$. The eigenvectors of $\mathbf{A}^H\mathbf{A}$ are collected by columns into an $N \times N$ unitary [2] matrix $\mathbf{V}$, while the eigenvectors of $\mathbf{A}\mathbf{A}^H$ are collected by columns in the unitary matrix $\mathbf{U}$, the singular values are collected in an $M \times N$ zero matrix $\mathbf{\Sigma}$ with main diagonal entries set to the singular values. Then

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H$$

is the singular value decomposition of $\mathbf{A}$. Every matrix has a singular value decomposition. Sometimes, the singular values are called the spectrum of $\mathbf{A}$. The command **linalg.svd** will return $\mathbf{U}$, $\mathbf{V}^H$, and σ_i as an array of the singular values. To obtain the matrix $\mathbf{\Sigma}$, use **linalg.diagsvd**. The following example illustrates the use of **linalg.svd**:

```
>>> import numpy as np
>>> from scipy import linalg
>>> A = np.array([[1,2,3],[4,5,6]])
>>> A
array([[1, 2, 3],
       [4, 5, 6]])
>>> M,N = A.shape
>>> U,s,Vh = linalg.svd(A)
>>> Sig = linalg.diagsvd(s,M,N)
>>> U, Vh = U, Vh
>>> U
array([[-0.3863177 , -0.92236578],
       [-0.92236578,  0.3863177 ]])
>>> Sig
array([[ 9.508032  ,  0.        ,  0.        ],
       [ 0.        ,  0.77286964,  0.        ]])
>>> Vh
array([[-0.42866713, -0.56630692, -0.7039467 ],
       [ 0.80596391,  0.11238241, -0.58119908],
       [ 0.40824829, -0.81649658,  0.40824829]])
>>> U.dot(Sig.dot(Vh)) #check computation
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
```

[1] A hermitian matrix $\mathbf{D}$ satisfies $\mathbf{D}^H = \mathbf{D}$.

[2] A unitary matrix $\mathbf{D}$ satisfies $\mathbf{D}^H\mathbf{D} = \mathbf{I} = \mathbf{D}\mathbf{D}^H$ so that $\mathbf{D}^{-1} = \mathbf{D}^H$.

LU decomposition

The LU decomposition finds a representation for the $M \times N$ matrix $\mathbf{A}$ as

$$\mathbf{A} = \mathbf{P}\,\mathbf{L}\,\mathbf{U},$$

where $\mathbf{P}$ is an $M \times M$ permutation matrix (a permutation of the rows of the identity matrix), $\mathbf{L}$ is in $M \times K$ lower triangular or trapezoidal matrix ($K = \mathbf{min}\,(M, N)$) with unit-diagonal, and $\mathbf{U}$ is an upper triangular or trapezoidal matrix. The SciPy command for this decomposition is `linalg.lu`.

Such a decomposition is often useful for solving many simultaneous equations where the left-hand side does not change but the right-hand side does. For example, suppose we are going to solve

$$\mathbf{A}\mathbf{x}_i = \mathbf{b}_i$$

for many different $\mathbf{b}_i$. The LU decomposition allows this to be written as

$$\mathbf{PLU}\mathbf{x}_i = \mathbf{b}_i.$$

Because $\mathbf{L}$ is lower-triangular, the equation can be solved for $\mathbf{U}\mathbf{x}_i$ and, finally, $\mathbf{x}_i$ very rapidly using forward- and back-substitution. An initial time spent factoring $\mathbf{A}$ allows for very rapid solution of similar systems of equations in the future. If the intent for performing LU decomposition is for solving linear systems, then the command `linalg.lu_factor` should be used followed by repeated applications of the command `linalg.lu_solve` to solve the system for each new right-hand side.

Cholesky decomposition

Cholesky decomposition is a special case of LU decomposition applicable to Hermitian positive definite matrices. When $\mathbf{A} = \mathbf{A}^H$ and $\mathbf{x}^H\mathbf{A}\mathbf{x} \geq 0$ for all $\mathbf{x}$, then decompositions of $\mathbf{A}$ can be found so that

$$\mathbf{A} = \mathbf{U}^H\mathbf{U}$$
$$\mathbf{A} = \mathbf{L}\mathbf{L}^H$$

,

where $\mathbf{L}$ is lower triangular and $\mathbf{U}$ is upper triangular. Notice that $\mathbf{L} = \mathbf{U}^H$**.** The command `linalg.cholesky` computes the Cholesky factorization. For using the Cholesky factorization to solve systems of equations, there are also `linalg.cho_factor` and `linalg.cho_solve` routines that work similarly to their LU decomposition counterparts.

QR decomposition

The QR decomposition (sometimes called a polar decomposition) works for any $M \times N$ array and finds an $M \times M$ unitary matrix $\mathbf{Q}$ and an $M \times N$ upper-trapezoidal matrix $\mathbf{R}$, such that

$$\mathbf{A} = \mathbf{Q}\mathbf{R}.$$

Notice that if the SVD of $\mathbf{A}$ is known, then the QR decomposition can be found.

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H = \mathbf{Q}\mathbf{R}$$

implies that $\mathbf{Q} = \mathbf{U}$ and $\mathbf{R} = \mathbf{\Sigma}\mathbf{V}^H$**.** Note, however, that in SciPy independent algorithms are used to find QR and SVD decompositions. The command for QR decomposition is `linalg.qr`.

Schur decomposition

For a square $N \times N$ matrix, $\mathbf{A}$, the Schur decomposition finds (not necessarily unique) matrices $\mathbf{T}$ and $\mathbf{Z}$, such that

$$\mathbf{A} = \mathbf{Z}\mathbf{T}\mathbf{Z}^H,$$

where $\mathbf{Z}$ is a unitary matrix and $\mathbf{T}$ is either upper triangular or quasi upper triangular, depending on whether or not a real Schur form or complex Schur form is requested. For a real Schur form both $\mathbf{T}$ and $\mathbf{Z}$ are real-valued when $\mathbf{A}$ is real-valued. When $\mathbf{A}$ is a real-valued matrix, the real Schur form is only quasi upper triangular because 2×2 blocks extrude from the main diagonal corresponding to any complex-valued eigenvalues. The command `linalg.schur` finds the Schur decomposition, while the command `linalg.rsf2csf` converts $\mathbf{T}$ and $\mathbf{Z}$ from a real Schur form to a complex Schur form. The Schur form is especially useful in calculating functions of matrices.

The following example illustrates the Schur decomposition:

```
>>> from scipy import linalg
>>> A = np.mat('[1 3 2; 1 4 5; 2 3 6]')
>>> T, Z = linalg.schur(A)
>>> T1, Z1 = linalg.schur(A, 'complex')
>>> T2, Z2 = linalg.rsf2csf(T, Z)
>>> T
array([[ 9.90012467,  1.78947961, -0.65498528],
       [ 0.        ,  0.54993766, -1.57754789],
       [ 0.        ,  0.51260928,  0.54993766]])
>>> T2
array([[ 9.90012467+0.00000000e+00j, -0.32436598+1.55463542e+00j,
        -0.88619748+5.69027615e-01j],
       [ 0.        +0.00000000e+00j,  0.54993766+8.99258408e-01j,
         1.06493862+3.05311332e-16j],
       [ 0.        +0.00000000e+00j,  0.        +0.00000000e+00j,
         0.54993766-8.99258408e-01j]])
>>> abs(T1 - T2) # different
array([[  1.06604538e-14,   2.06969555e+00,   1.69375747e+00],  # may vary
       [  0.00000000e+00,   1.33688556e-15,   4.74146496e-01],
       [  0.00000000e+00,   0.00000000e+00,   1.13220977e-15]])
>>> abs(Z1 - Z2) # different
array([[ 0.06833781,  0.88091091,  0.79568503],    # may vary
       [ 0.11857169,  0.44491892,  0.99594171],
       [ 0.12624999,  0.60264117,  0.77257633]])
>>> T, Z, T1, Z1, T2, Z2 = map(np.mat,(T,Z,T1,Z1,T2,Z2))
>>> abs(A - Z*T*Z.H)  # same
matrix([[  5.55111512e-16,   1.77635684e-15,   2.22044605e-15],
        [  0.00000000e+00,   3.99680289e-15,   8.88178420e-16],
        [  1.11022302e-15,   4.44089210e-16,   3.55271368e-15]])
>>> abs(A - Z1*T1*Z1.H)  # same
matrix([[  4.26993904e-15,   6.21793362e-15,   8.00007092e-15],
        [  5.77945386e-15,   6.21798014e-15,   1.06653681e-14],
        [  7.16681444e-15,   8.90271058e-15,   1.77635764e-14]])
>>> abs(A - Z2*T2*Z2.H)  # same
matrix([[  6.02594127e-16,   1.77648931e-15,   2.22506907e-15],
        [  2.46275555e-16,   3.99684548e-15,   8.91642616e-16],
        [  8.88225111e-16,   8.88312432e-16,   4.44104848e-15]])
```

Interpolative decomposition

`scipy.linalg.interpolative` contains routines for computing the interpolative decomposition (ID) of a matrix. For a matrix $A \in \mathbb{C}^{m \times n}$ of rank $k \leq \min\{m, n\}$ this is a factorization

$$A\Pi = \begin{bmatrix} A\Pi_1 & A\Pi_2 \end{bmatrix} = A\Pi_1 \begin{bmatrix} I & T \end{bmatrix},$$

where $\Pi = [\Pi_1, \Pi_2]$ is a permutation matrix with $\Pi_1 \in \{0, 1\}^{n \times k}$, i.e., $A\Pi_2 = A\Pi_1 T$. This can equivalently be written as $A = BP$, where $B = A\Pi_1$ and $P = [I, T]\Pi^{\mathsf{T}}$ are the *skeleton* and *interpolation matrices*, respectively.

> **See also**
>
> `scipy.linalg.interpolative` — for more information.

Matrix functions

Consider the function $f(x)$ with Taylor series expansion

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(0)}{k!} x^k.$$

A matrix function can be defined using this Taylor series for the square matrix $\mathbf{A}$ as

$$f(\mathbf{A}) = \sum_{k=0}^{\infty} \frac{f^{(k)}(0)}{k!} \mathbf{A}^k.$$

> **Note**
>
> While this serves as a useful representation of a matrix function, it is rarely the best way to calculate a matrix function. In particular, if the matrix is not diagonalizable, results may be innacurate.

Exponential and logarithm functions

The matrix exponential is one of the more common matrix functions. The preferred method for implementing the matrix exponential is to use scaling and a Padé approximation for e^x. This algorithm is implemented as `linalg.expm`.

The inverse of the matrix exponential is the matrix logarithm defined as the inverse of the matrix exponential:

$$\mathbf{A} \equiv \exp(\log(\mathbf{A})).$$

The matrix logarithm can be obtained with `linalg.logm`.

Trigonometric functions

The trigonometric functions, $\sin$, $\cos$, and $\tan$, are implemented for matrices in `linalg.sinm`, `linalg.cosm`, and `linalg.tanm`, respectively. The matrix sine and cosine can be defined using Euler's identity as

$$\begin{aligned}\sin(\mathbf{A}) &= \frac{e^{j\mathbf{A}} - e^{-j\mathbf{A}}}{2j} \\ \cos(\mathbf{A}) &= \frac{e^{j\mathbf{A}} + e^{-j\mathbf{A}}}{2}.\end{aligned}$$

The tangent is

$$\tan(x) = \frac{\sin(x)}{\cos(x)} = [\cos(x)]^{-1}\sin(x)$$

and so the matrix tangent is defined as

$$[\cos(\mathbf{A})]^{-1}\sin(\mathbf{A}).$$

Hyperbolic trigonometric functions

The hyperbolic trigonometric functions, $\sinh$, $\cosh$, and $\tanh$, can also be defined for matrices using the familiar definitions:

$$\begin{aligned}\sinh(\mathbf{A}) &= \frac{e^{\mathbf{A}} - e^{-\mathbf{A}}}{2} \\ \cosh(\mathbf{A}) &= \frac{e^{\mathbf{A}} + e^{-\mathbf{A}}}{2} \\ \tanh(\mathbf{A}) &= [\cosh(\mathbf{A})]^{-1}\sinh(\mathbf{A}).\end{aligned}$$

These matrix functions can be found using `linalg.sinhm`, `linalg.coshm`, and `linalg.tanhm`.

Arbitrary function

Finally, any arbitrary function that takes one complex number and returns a complex number can be called as a matrix function using the command `linalg.funm`. This command takes the matrix and an arbitrary Python function. It then implements an algorithm from Golub and Van Loan's book "Matrix Computations" to compute the function applied to the matrix using a Schur decomposition. Note that *the function needs to accept complex numbers* as input in order to work with this algorithm. For example, the following code computes the zeroth-order Bessel function applied to a matrix.

```
>>> from scipy import special, linalg
>>> rng = np.random.default_rng()
>>> A = rng.random((3, 3))
>>> B = linalg.funm(A, lambda x: special.jv(0, x))
>>> A
array([[0.06369197, 0.90647174, 0.98024544],
       [0.68752227, 0.5604377 , 0.49142032],
       [0.86754578, 0.9746787 , 0.37932682]])
>>> B
array([[ 0.6929219 , -0.29728805, -0.15930896],
       [-0.16226043,  0.71967826, -0.22709386],
       [-0.19945564, -0.33379957,  0.70259022]])
>>> linalg.eigvals(A)
array([ 1.94835336+0.j, -0.72219681+0.j, -0.22270006+0.j])
>>> special.jv(0, linalg.eigvals(A))
array([0.25375345+0.j, 0.87379738+0.j, 0.98763955+0.j])
>>> linalg.eigvals(B)
array([0.25375345+0.j, 0.87379738+0.j, 0.98763955+0.j])
```

Note how, by virtue of how matrix analytic functions are defined, the Bessel function has acted on the matrix eigenvalues.

Special matrices

SciPy and NumPy provide several functions for creating special matrices that are frequently used in engineering and science.

Type	Function	Description
block diagonal	`scipy.linalg.block_diag`	Create a block diagonal matrix from the provided arrays.
circulant	`scipy.linalg.circulant`	Create a circulant matrix.
companion	`scipy.linalg.companion`	Create a companion matrix.
convolution	`scipy.linalg.convolution_matrix`	Create a convolution matrix.
Discrete Fourier	`scipy.linalg.dft`	Create a discrete Fourier transform matrix.
Fiedler	`scipy.linalg.fiedler`	Create a symmetric Fiedler matrix.
Fiedler Companion	`scipy.linalg.fiedler_companion`	Create a Fiedler companion matrix.
Hadamard	`scipy.linalg.hadamard`	Create an Hadamard matrix.
Hankel	`scipy.linalg.hankel`	Create a Hankel matrix.
Helmert	`scipy.linalg.helmert`	Create a Helmert matrix.
Hilbert	`scipy.linalg.hilbert`	Create a Hilbert matrix.
Inverse Hilbert	`scipy.linalg.invhilbert`	Create the inverse of a Hilbert matrix.
Leslie	`scipy.linalg.leslie`	Create a Leslie matrix.
Pascal	`scipy.linalg.pascal`	Create a Pascal matrix.
Inverse Pascal	`scipy.linalg.invpascal`	Create the inverse of a Pascal matrix.
Toeplitz	`scipy.linalg.toeplitz`	Create a Toeplitz matrix.
Van der Monde	`numpy.vander`	Create a Van der Monde matrix.

For examples of the use of these functions, see their respective docstrings.

Previous
Signal Processing (`scipy.signal`)

Next
Sparse Arrays (`scipy.sparse`)

Created using Sphinx 5.3.0.