2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

3
SciPy for Linear Algebra

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

In the following chapters, we will continue exploring the different SciPy modules through meaningful examples. We will start with the treatment of matrices (whether normal or sparse) with the modules on Linear Algebra – `linalg` and `sparse.linalg` – which expand and improve the NumPy module with the same name.

This discipline of mathematics mainly studies vector spaces and the linear mappings among them. Matrices represent objects in this field naturally, in such a way that any property of the underlying objects may be obtained by performing some operation on the representing matrices. We assume at this point that you are familiar with at least the basics of linear algebra, in particular with the notion of matrix multiplication, finding the determinant and inverse of a matrix, as well as their immediate applications in vector calculus.

Matrix creation

In SciPy, a matrix structure is given to any one- or two-dimensional `ndarray`, with either the `matrix` or `mat` command. The complete syntax is as follows:

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

```
numpy.matrix(data=object, dtype=None, copy=True)
```

In the creation of matrices, the data may be given as a string or as `ndarray`, which is very convenient. When using strings, the semicolon denotes change of row, and the comma denotes change of column:

```
>>> A=numpy.matrix("1,2,3;4,5,6")
>>> A
matrix([[1, 2, 3],
        [4, 5, 6]])
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 50
http://site.ebrary.com/id/10672339?ppg=50
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary


```
>>> A=numpy.matrix([[1,2,3],[4,5,6]])
>>> A
matrix([[1, 2, 3],
        [4, 5, 6]])
```

Another way of creating a matrix from a two-dimensional array is by enforcing the matrix structure on a new object, copying the data of the former with the `asmatrix` routine.

We say that a matrix is sparse if most of its entries are zeros. It is a waste of memory to input such matrices in the usual way, especially if the dimensions are large, and SciPy contemplates different procedures to store such matrices effectively in memory. Most of the usual methods to input sparse matrices are contemplated in SciPy as routines in the `scipy.sparse` module. Some of those methods are block sparse row (`bsr_matrix`), coordinate format (`coo_matrix`), compressed sparse column or row (`csc_matrix`, `csr_matrix`), sparse matrix with diagonal storage (`dia_matrix`), dictionary with keys-based sorting (`dok_matrix`), and row-based linked list (`lil_matrix`).

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

At this point, we would like to present at least one of them: the coordinate format. In this format, given a sparse matrix `A`, we identify the coordinates of the nonzero elements, say n of them, and we create two n-dimensional `ndarray` arrays containing in order, the columns and rows of those entries, and a third `ndarray` containing the values of the corresponding entries. For instance, notice the following sparse matrix:

$$\begin{pmatrix} 0 & 10 & 0 & 0 & 0 \\ 0 & 0 & 20 & 0 & 0 \\ 0 & 0 & 0 & 30 & 0 \\ 0 & 0 & 0 & 0 & 40 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

One of the nonzero entries is at the second column and first row (this is the location (1, 0) in Python) and has the value, 10. Another nonzero entry is at (2, 1) and has the value, 20. A third nonzero entry, with the value 30, is located at (3, 2). The last nonzero entry of `A` is located at (4, 3), and has the value, 40.

We then have `ndarray` of rows, another `ndarray` of columns, and another `ndarray` of values:

```
>>> rows=numpy.array([0,1,2,3])
>>> cols=numpy.array([1,2,3,4])
>>> vals=numpy.array([10,20,30,40])
```


2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 51
http://site.ebrary.com/id/10672339?ppg=51
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

We create the matrix `A` as follows:

```
>>> import scipy.sparse
>>> A=scipy.sparse.coo_matrix( (vals,(rows,cols)) )
>>> print A; print A.todense()
  (0, 1)  10.0
  (1, 2)  20.0
  (2, 3)  30.0
  (3, 4)  40.0
[[  0.  10.   0.   0.   0.]
 [  0.   0.  20.   0.   0.]
 [  0.   0.   0.  30.   0.]
 [  0.   0.   0.   0.  40.]]
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Notice how the `todense` method turns sparse matrices into full matrices. Also note that it obviates any row or column of full zeros following the last nonzero element.

Associated to each input method, we have functions that identify sparse matrices of each kind. For instance, if we suspect that `A` is a sparse matrix in the `coo_matrix` format, we may use the following command:

```
>>> scipy.sparse.isspmatrix_coo(A)
True
```

All the array routines cast to matrices, provided the input is a matrix. This is very convenient for matrix creation, especially thanks to stacking commands (`hstack`, `vstack`, `tile`). Besides these, matrices enjoy one more amazing stacking command, `bmat`. This routine allows the stacking of matrices by means of strings, making use of the convention "semicolon for change of row, comma for change of column", and allowing matrix names inside of the string to be evaluated. The following example is enlightening:

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

```
>>> B=numpy.mat(numpy.ones((3,3)))
>>> W=numpy.mat(numpy.zeros((3,3)))
>>> print numpy.bmat('B,W;W,B')
[[ 1.  1.  1.  0.  0.  0.]
 [ 1.  1.  1.  0.  0.  0.]
 [ 1.  1.  1.  0.  0.  0.]
 [ 0.  0.  0.  1.  1.  1.]
 [ 0.  0.  0.  1.  1.  1.]
 [ 0.  0.  0.  1.  1.  1.]]
```


2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 52
http://site.ebrary.com/id/10672339?ppg=52
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

The main difference between arrays and matrices is in regards to the behavior of the product of two objects of the same type. For example, multiplication between two arrays means "element-wise multiplication of the entries of the two arrays", and requires two objects of the same shape.

```
>>> a=numpy.array([[1,2],[3,4]])
>>> a*a
array([[ 1,  4],
       [ 9, 16]])
```

On the other hand, matrix multiplication requires a first matrix with shape (m, n), and a second matrix with shape (n, p)—the number of columns in the first matrix must be the same as the number of rows in the second matrix. This operation offers a new matrix of shape (m, p), as shown in the following diagram:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 7 & 10 \\ 15 & 22 \end{pmatrix}$$

The following is the code snippet:

```
>>> A=numpy.mat(a)
>>> A*A
matrix([[ 7, 10],
        [15, 22]])
```

If we desire to perform an element-wise multiplication of the elements of two matrices, we may do so with the versatile `numpy.multiply` command, as follows:

```
>>> numpy.multiply(A,A)
matrix([[ 1,  4],
        [ 9, 16]])
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

The other notable difference between arrays and matrices is in regards to their shapes. While we allow one-dimensional arrays, their corresponding matrices must have two dimensions. This is very important to have in mind when we transpose either object.

```
>>> a=numpy.arange(5); A=numpy.mat(a)
>>> a.shape, A.shape, a.transpose().shape, A.transpose().shape
((5,), (1, 5), (5,), (5, 1))
```


2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 53
http://site.ebrary.com/id/10672339?ppg=53
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

SciPy extends the basic applications that we access by offering interesting matrix creation commands, and many related methods. It also allows us the opportunity to speed up computations in the cases where special matrices are used.

The `scipy.linalg` module allows the creation of the special matrices such as, block diagonal matrices from provided arrays (`block_diag`), circulant matrices (`circulant`), companion matrices (`companion`), Hadamard matrices (`hadamard`), Hankel matrices (`hankel`), Hilbert and inverse Hilbert matrices (`hilbert`, `invhilbert`), Leslie matrices (`leslie`), square Pascal matrices (`pascal`), Toeplitz matrices (`toeplitz`), and lower-triangular matrices (`tri`).

Let's see an example on optimal weightings.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Suppose we are given p objects to be weighed in n weighings with a two-pan balance. We create an n x p matrix of plus-minus ones, where a positive value in the position (i, j) indicates that the j^{th} object is placed in the left pan of the balance in the i^{th} weighing, and a negative value indicates that the j^{th} object is placed in the right pan of the balance in the i^{th} weighing.

It is known that optimal weighings are designed by submatrices of Hadamard matrices. For the problem of designing an optimal weighing for eight objects with three weighings, we could then explore different choices of three rows of a Hadamard matrix of order eight. The only requirement is that the sum of the elements on the row of the matrix is zero (so that the same number of objects is placed on each pan). With some smart slicing, we can accomplish just that:

```
>>> A=scipy.linalg.hadamard(8)
>>> zero_sum_rows = (numpy.sum(A,0)==0)
>>> B=A[zero_sum_rows,:]
>>> print B[0:3,:]
[[ 1 -1  1 -1  1 -1  1 -1]
 [ 1  1 -1 -1  1  1 -1 -1]
 [ 1 -1 -1  1  1 -1 -1  1]]
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

The `scipy.sparse` module has its own set of special matrices. The most common are matrices of ones along diagonals (`eye`), identity matrices (`identity`), matrices from diagonals (`diags`, `spdiags`), block diagonal matrices from sparse matrices (`block_diag`), matrices from sparse sub-blocks (`bmat`), column-wise and row-wise stacks (`hstack`, `vstack`), and random matrices of given shape and density with uniformly distributed values (`rand`).

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 54
http://site.ebrary.com/id/10672339?ppg=54
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Matrix methods

Besides inheriting all the array methods, matrices enjoy four extra attributes – `T` for transpose, `H` for conjugate transpose, `I` for inverse, and `A` to cast as `ndarray`.

```
>>> A = numpy.matrix("1+1j, 2-1j; 3-1j, 4+1j")
>>> print A.T; print A.H
[[ 1.+1.j  3.-1.j]
 [ 2.-1.j  4.+1.j]]
[[ 1.-1.j  3.+1.j]
 [ 2.+1.j  4.-1.j]]
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Operations between matrices

We have briefly covered the most basic operation between two matrices, the matrix product. For any other kind of product we resort to the basic utilities in the NumPy libraries – dot product for arrays or vectors (`dot`, `vdot`), inner and outer products of two arrays (`inner`, `outer`), tensor dot product along specified axes (`tensordot`), or the Kronecker product of two arrays (`kron`).

Let's see an example on creation of orthonormal bases.

Create an orthonormal basis of the nine-dimensional real space from an orthonormal basis of the three-dimensional real space.

For example, we choose the orthonormal basis formed by the vectors.

$$\begin{aligned} v_1 &= \tfrac{1}{\sqrt{2}}(1,0,1), \\ v_2 &= (0,1,0), \\ v_3 &= \tfrac{1}{\sqrt{2}}(1,0,-1) \end{aligned}$$

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

We compute the desired basis by collecting these vectors in a matrix and using a Kronecker product, as follows:

```
>>> mu=1/numpy.sqrt(2)
>>> A=numpy.matrix([[mu,0,mu],[0,1,0],[mu,0,-mu]])
>>> B=scipy.linalg.kron(A,A)
```


2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 55
http://site.ebrary.com/id/10672339?ppg=55
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

The columns of the matrix `B` shown previously, give us an orthonormal basis directly. For instance, the vectors with odd indices would be the columns of the following submatrix:

```
>>> print B[:,0:-1:2]
[[ 0.5  0.5  0.   0.5]
 [ 0.   0.   0.   0. ]
 [ 0.5 -0.5  0.   0.5]
 [ 0.   0.   0.   0. ]
 [ 0.   0.   1.   0. ]
 [ 0.  -0.   0.   0. ]
 [ 0.5  0.5  0.  -0.5]
 [ 0.   0.   0.  -0. ]
 [ 0.5 -0.5  0.  -0.5]]
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Functions on matrices

The `scipy.linalg` module offers a useful set of functions on matrices. The basic two commands on square matrices are `inv` (for the inverse of a matrix) and `det` (for the determinant). The power of a square matrix is given by the normal exponentiation; that is, if `A` is a square matrix, then `A**2` indicates the matrix product `A*A`.

```
>>> A=numpy.matrix("1,1j;21,3")
>>> print A**2; print numpy.asarray(A)**2
[[-1.+0.j  0.+4.j]
 [ 0.+8.j  7.+0.j]]
[[ 1.+0.j -1.+0.j]
 [-4.+0.j  9.+0.j]]
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

More advanced commands compute matrix functions that rely on power series representation of expressions involving matrix powers, such as the matrix exponential (for which there are three possibilities – `expm`, `expm2`, and `expm3`), the matrix logarithm (`logm`), matrix trigonometric functions (`cosm`, `sinm`, `tanm`), matrix hyperbolic trigonometric functions (`coshm`, `sinhm`, `tanhm`), the matrix sign function (`signm`), or the matrix square root (`sqrtm`).

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 56
http://site.ebrary.com/id/10672339?ppg=56
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Notice the difference between the application of the normal exponential function on a matrix, and the result of a matrix exponential function. In the former case, we obtain the application of `numpy.exp` to each entry of the matrix; in the latter, we actually compute the exponential of the matrix following the power series representation:

$$e^A = \sum_{n=0}^{\infty} \frac{1}{n!} A^n$$

The following is the code snippet:

```
>>> a=numpy.arange(0,2*numpy.pi,1.6)
>>> A = scipy.linalg.toeplitz(a)
>>> print A
[[ 0.   1.6  3.2  4.8]
 [ 1.6  0.   1.6  3.2]
 [ 3.2  1.6  0.   1.6]
 [ 4.8  3.2  1.6  0. ]]
>>> print numpy.exp(A)
[[   1.           4.95303242   24.5325302   121.51041752]
 [   4.95303242    1.           4.95303242   24.5325302 ]
 [  24.5325302     4.95303242    1.           4.95303242]
 [ 121.51041752   24.5325302     4.95303242    1.        ]]
>>> print scipy.linalg.expm(A)
[[ 1271.76972856   916.49316549   916.63015271  1271.70874469]
 [  916.49316549   660.86560972   660.5306514    916.63015271]
 [  916.63015271   660.5306514    660.86560972   916.49316549]
 [ 1271.70874469   916.63015271   916.49316549  1271.76972856]]
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

For sparse square matrices, we have an optimized inverse function, as well as a matrix exponential – `scipy.sparse.linalg.inv`, `scipy.sparse.linalg.expm`.

For general matrices, we have the basic norm function (`norm`), as well as two versions of the Moore-Penrose pseudoinverse (`pinv` and `pinv2`).

Once again, we need to emphasize how important it is to rely on these functions, rather than coding their equivalent expressions manually. For instance, note the norm computation of vectors or matrices, `scipy.linalg.norm`. Let us show, by example, the two-norm of a two-dimensional vector `v=numpy.matrix([x,y])`, where at least one of the `x` and `y` values is extremely large – large enough so that `x*x` overflows.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 57
http://site.ebrary.com/id/10672339?ppg=57
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary


```
>>> x=10**100; y=9; v=numpy.matrix([x,y])
>>> scipy.linalg.norm(v,2)    # the right method
9.9999999999999982e+99
>>> numpy.sqrt(x*x+y*y)       # the wrong method
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: sqrt
```

Eigenvalue problems and matrix decompositions

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Another set of operations required on matrices is related to the computation and handling of eigenvalues and eigenvectors of square matrices. These two problems rank among the most complex operations that we can perform on square matrices, and extensive research has been put to obtaining good algorithms with low complexity and optimal usage of memory resources. Scipy has state-of-the-art code for implementing these ideas.

For the computation of eigenvalues, the `scipy.linalg` module provides with the three routines, such as `eigvals` (for any ordinary or general eigenvalue problem), `eigvalsh` (if the matrix is symmetric of complex Hermitian), and `eigvals_banded` (if the matrix is banded). To compute the eigenvectors, we also have three corresponding choices – `eig`, `eigh`, and `eigh_banded`.

The syntax in all cases is very similar. For example, for the general case of eigenvalues, we use the following line of code:

```
eigvals(A, B=None, overwrite_a=False)
```

The matrix `A` must be square, of course. It should be the only parameter passed to the routine if we wish to solve an ordinary eigenvalue problem. If we wish to generalize it, we may provide with an extra square matrix (of the same dimensions as matrix `A`). This is passed in the `B` parameter.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

The module also offers an extensive collection of functions that compute different decompositions of matrices, as follows:

- **Pivoted LU decomposition**: We can use the `lu` and `lufactor` commands.
- **Singular value decomposition**: We can use the `svd` command. To compute the singular values, we issue `svdvals`. If we wish to compose the sigma matrix in the singular value decomposition from its singular values, we do so with the `diagsvd` routine. If we wish to compute an orthogonal basis for the range of a matrix using SVD, we can do so with the `orth` command.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 58
http://site.ebrary.com/id/10672339?ppg=58
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

- **Cholesky decomposition**: We can use `cholesky`, `cholesky_banded`, `cho_factor`.
- **QR and QZ decompositions**: We can use the `qr` and `qz` commands. If we wish to multiply a matrix with the matrix Q of a decomposition, we use the syntactic sugar `qr_multiply`, rather than performing this procedure in two steps.
- **Schur and Hessenberg decompositions**: We can use `schur` and `Hessenberg`. If we wish to convert a real Schur form to complex, we have the `rsf2csf` routine.

At this point we have an interesting application, which makes use of some of the routines explained so far, image compression.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Image compression via the singular value decomposition

This is a very simple application where a square image `A` of size n x n, stored as `ndarray` is regarded as a matrix, and **singular value decomposition** (**SVD**) is performed on it.

$$A = U \cdot S \cdot V^*, \quad U = \begin{pmatrix} u_1 \\ \vdots \\ u_n \end{pmatrix}, S = \begin{pmatrix} s_1 & & \\ & \ddots & \\ & & s_n \end{pmatrix}, V^* = \begin{pmatrix} v_1 & \cdots & v_n \end{pmatrix}$$

From all the singular values of s we choose a fraction, together with their corresponding left and right singular vectors u, v. We compute a new matrix by collecting them according to the formula given in the following diagram:

$$\sum_{j=1}^{k} s_j \left(u_j \cdot v_j \right)$$

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Note, for example, how much alike are the original (512 singular values) and an approximation using only 32 singular values:

```
import scipy
from scipy.linalg import svd
import matplotlib.pyplot as plt
img=scipy.misc.lena()
U,s,Vh=svd(img)      # Singular Value Decomposition
A = numpy.dot( U[:,0:32],  # use only 32 singular values
```

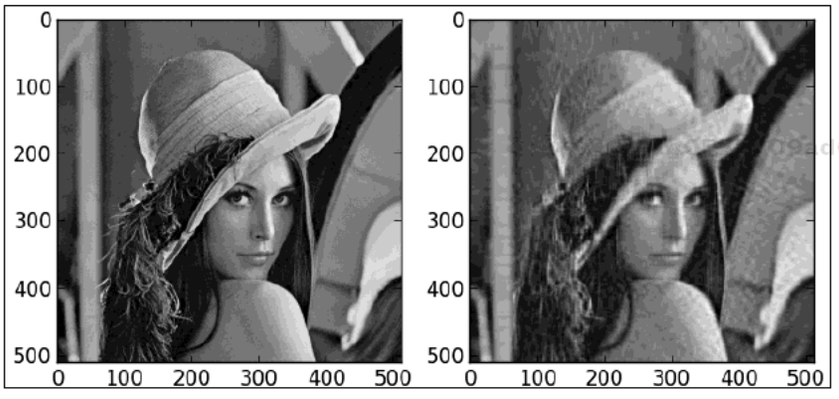

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 59
http://site.ebrary.com/id/10672339?ppg=59
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary


```
            numpy.dot( numpy.diag(s[0:32]),
                       Vh[0:32,:]))
plt.subplot(121,aspect='equal'); plt.imshow(img); plt.gray()
plt.subplot(122,aspect='equal'); plt.imshow(A)
```

This produces the following images, of which the left one is the original image and the right one shows the approximation via 32 singular values:



The obvious advantage comes upon the realization that for the full image we need 512 times 512 coefficients (that is 262,144 floating point units), whereas for this approximation via SVD, we only need 32,800 (*(2 * 32 * 512) + 32*) coefficients. This is one-eighth of the original information.

Solvers

One of the main applications of linear algebra is to the solution of large systems of linear equations. For the basic systems of the form `Ax=b`, for any square matrix `A` and a general matrix `b` (with as many rows as columns in `A`), we have two generic methods to find `x` (`solve` for dense matrices and `spsolve` for sparse matrices), with the following syntax:

```
solve(A, b, sym_pos=False, lower=False, overwrite_a=False, overwrite_
b=False, debug=False)
spsolve(A, b[, permc_spec, use_umfpack])
```

There are solvers that are more sophisticated in SciPy, with enhanced performance for situations in which the structure of the matrix `A` is known. For dense matrices we have three commands in the `scipy.linalg` module – `solve_banded` (for banded matrices), `solveh_banded` (if besides banded, `A` is Hermitian), and `solve_triangular` (for triangular matrices).

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 60
http://site.ebrary.com/id/10672339?ppg=60
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

When a solution is not possible (for example, if `A` is a singular matrix), it is still possible to obtain a matrix `x` that minimizes the norm of `b-Ax` in the least-squares sense. We can compute such a matrix with the `lstsq` command, which has the following syntax:

```
lstsq(A, b, cond=None, overwrite_a=False, overwrite_b=False)
```

The output of this function is a tuple that contains the following:

- The solution found (as `ndarray`)
- The sum of residues (as another `ndarray`)
- The effective rank of the matrix `A`
- The singular values of the matrix `A` (as another `ndarray`)

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Let us illustrate this routine with a simple example, to solve the following system:

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

The following is the code snippet:

```
>>> A=numpy.mat(numpy.eye(3,k=1))
>>> b=numpy.mat(numpy.arange(3)).T
>>> xinfo=scipy.linalg.lstsq(A,b)
>>> print xinfo[0].T      # output the solution
[[ 0.  0.  1.]]
```

The `overwrite_` options are designed to enhance performance of the algorithms, and should be used carefully, since they destroy the original data.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

The truly fastest solvers coded in SciPy are based upon decomposition of matrices. Reducing the system into something simpler easily solves huge and really complicated systems of linear equations. We may do so at this point using the decompositions presented in the previous section, but of course the SciPy philosophy is to help us deal with all the nuisances of memory and resources internally. We then have the extra solvers coded in this module, such as `lu_solve` (for solutions based on LU decompositions), and `cho_solve`, `cho_solve_banded` (for solutions based on Cholesky decompositions).

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 61
http://site.ebrary.com/id/10672339?ppg=61
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

There are also solvers for more complex matrix equations – the Sylvester equation (`solve_sylvester`), both the continuous and discrete algebraic Riccati equations (`solve_continuous_are`, `solve_discrete_are`); and both the continuous and discrete Lyapunov equations (`solve_discrete_lyapunov`, `solve_lyapunov`).

Most of the matrix decompositions and solutions to eigenvalue problems are contemplated for sparse matrices in the `scipy.sparse.linalg` module, with a similar naming convention but much more robust use of computer resources and error control.

Summary

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

This chapter explored the treatment of matrices (whether normal or sparse) with the modules on linear algebra – `linalg` and `sparse.linalg`, which expand and improve the NumPy module with the same name.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 62
http://site.ebrary.com/id/10672339?ppg=62
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 63
http://site.ebrary.com/id/10672339?ppg=63
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.