Vectors and Matrices

Computational Methods, Oct. 2017, Kenji Doya

Here we work with vectors and matrices, and get acquainted with concepts in linear algebra by computing.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Numpy *ndarray* as vector and matrix

You can create a matrix from a nested list.

In [2]:
```
A = np.array([[1, 2, 3], [4, 5, 6]])
A
```

Out[2]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

You cam also create a matrix by stacking arrays vertically

In [3]:
```
b = np.array([1, 2, 3])
np.vstack((b, 2*b))
```

Out[3]:
```
array([[1, 2, 3],
       [2, 4, 6]])
```

In [4]:
```
# another way of stacking row vectors
np.r_[[b, 2*b]]
```

Out[4]:
```
array([[1, 2, 3],
       [2, 4, 6]])
```

or by combining arrays as column vectors

In [5]:
```
np.c_[b, 2*b]
```

Out[5]:
```
array([[1, 2],
       [2, 4],
       [3, 6]])
```

Here are functions and attributes to check the type and shape of a vector or matrix.

In [6]:
```
A.dtype
```

Out[6]:
```
dtype('int64')
```

In [7]:
```
A.ndim
```

Out[7]:
```
2
```

In [8]:
```
A.shape
```

Out[8]:
```
(2, 3)
```

In [9]:
```
A.size
```

Out[9]:
```
6
```

In [10]:
```
len(A)
```

Out[10]:
```
2
```

A 2D numpy array is internally a linear sequence of data.
`ravel( )` geves its flattened representation.

In [11]:
```
A.ravel()
```

Out[11]:
```
array([1, 2, 3, 4, 5, 6])
```

The result of reshaping reflect this internal sequence.

In [12]:
```
A.reshape(3, 2)
```

Out[12]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

In [ ]:
```

```

adressing components by [ ]

In [13]:
```
A
```

Out[13]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [14]:
```
A[1]  # second row
```

Out[14]:
```
array([4, 5, 6])
```

In [15]:
```
A[:,1]  # second column
```

Out[15]:
```
array([2, 5])
```

In [16]:
```
A[0, [2,1,2,0]]  # first row, in a new order
```

Out[16]:
```
array([3, 2, 3, 1])
```

In [17]:
```
A[[1,0,1], [2,1,0]]   # [A[1,2], A[0,1], A[1,0]]
```

Out[17]:
```
array([6, 2, 4])
```

In [18]:
```
A[:,::-1]  # columns in reverse order
```

Out[18]:
```
array([[3, 2, 1],
       [6, 5, 4]])
```

In [ ]:
```

```

common matrices

In [19]:
```
np.zeros((2,3))
```

Out[19]:
```
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

In [20]:
```
np.ones((2,3))*3
```

Out[20]:
```
array([[ 3.,  3.,  3.],
       [ 3.,  3.,  3.]])
```

In [21]:
```
np.eye(3)
```

Out[21]:
```
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

In [22]:
```
np.arange(3, 6)
```

Out[22]:
```
array([3, 4, 5])
```

In [23]:
```
# any function by list comprehension
np.array([10*i + j for i in range(2) for j in range(3)]).reshape((2,3)
```

Out[23]:
```
array([[ 0,  1,  2],
       [10, 11, 12]])
```

In [24]:
```
A = np.zeros((2, 3))
for i in range(2):
    for j in range(3):
        A[i, j] = 10*i + j
print(A)
```

```
[[  0.   1.   2.]
 [ 10.  11.  12.]]
```

In [25]:
```
np.random.randn(2, 3, 2)
# try also randn
```

Out[25]:
```
array([[[-0.62834612,  1.06576917],
        [-0.07918368, -0.22060767],
        [ 1.07715479, -0.05428495]],

       [[ 0.33889854,  0.11252157],
        [ 0.60106908, -0.2353379 ],
        [ 0.429524  , -0.90284573]]])
```

In [26]:
```
np.random.randint(-5, 5, size=(2,3))
```

Out[26]:
```
array([[ 0, -4, -2],
       [-5, -1, -5]])
```

In [27]:
```
np.empty((2,3))  # the initial content is undetermined
```

Out[27]:
```
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

In [ ]:
```

```

Vectors and dot product

A vector can represent:

- a point in n-dimensional space
- a movement in n-dimensional space

The dot product (or inner product) of two vectors $\boldsymbol{x} = (x_1, \ldots, x_n)$ and $\boldsymbol{y} = (y_1, \ldots, y_n)$ is defined as

$$\boldsymbol{x} \cdot \boldsymbol{y} = x_1 y_1 + \ldots + x_n y_n = \sum_{i=1}^{n} x_i y_i$$

The inner product measures how two vectors match up, giving

$$-||\boldsymbol{x}||\,||\boldsymbol{y}|| \leq \boldsymbol{x} \cdot \boldsymbol{y} \leq ||\boldsymbol{x}||\,||\boldsymbol{y}||$$

with the maximum when two vectors line up, zero when two are orthogonal.

The length (or norm) of the vector is defined as

$$||\boldsymbol{x}|| = \sqrt{\sum_{i=1}^{n} x_i^2} = \sqrt{\boldsymbol{x} \cdot \boldsymbol{x}}$$

In [28]:
```
x = np.array([0, 1, 2])
y = np.array([3, 4, 5])
print( x * y)
print( np.inner(x, y))
print( np.dot(x, y))
print( x.dot(y))
print( x @ y)
```

```
[ 0  4 10]
14
14
14
14
```

In [ ]:

Matrices and matrix product

A matrix can prepresent

- a set of vectors
- time series of vectors
- 2D image data,
- ...

The matrix product AB is a matrix made of the inner products of the rows of A and the columns of B.

For $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ and $B = \begin{pmatrix} e & f \\ g & h \end{pmatrix}$, the matrix product is

$$AB = \begin{pmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{pmatrix}.$$

In [29]:
```
A = np.array([[0, 1], [2, 3]])
print(A)
B = np.array([[1, 2], [3, 4]])
print(B)
```

```
[[0 1]
 [2 3]]
[[1 2]
 [3 4]]
```

In [30]:
```
print( A * B)  # component-wise
print( np.inner(A, B))  # row by row
print( np.dot(A, B))
print( A.dot(B))
print( A @ B)  # new since Pyton 3.5
```

```
[[ 0  2]
 [ 6 12]]
[[ 2  4]
 [ 8 18]]
[[ 3  4]
 [11 16]]
[[ 3  4]
 [11 16]]
[[ 3  4]
 [11 16]]
```

You can explicitly define an array as a matrix object so that `*` works as a matrix product.

In [31]:
```
AA = np.matrix(A)
BB = np.matrix(B)
print( AA * BB)
print( AA * B)  # if one is a matrix, * is taken as matrix product
print( A * BB)
```

```
[[ 3  4]
 [11 16]]
[[ 3  4]
 [11 16]]
[[ 3  4]
 [11 16]]
```

And ** works as a matrix power

In [32]:
```
print( AA ** 2)
print( AA ** 3)
```

```
[[ 2  3]
 [ 6 11]]
[[ 6 11]
 [22 39]]
```

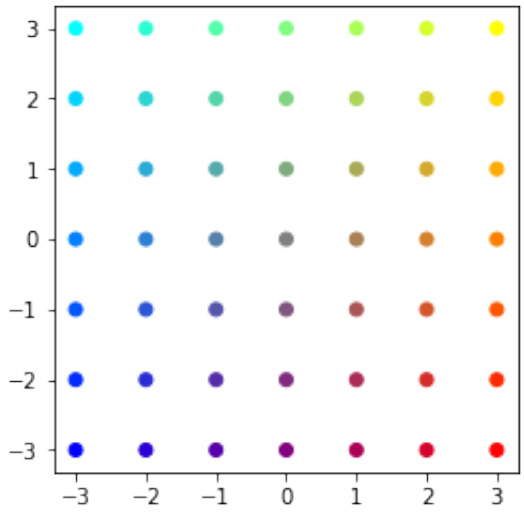
In [ ]:
```

```

Matrix and vector space

A matrix product can mean:

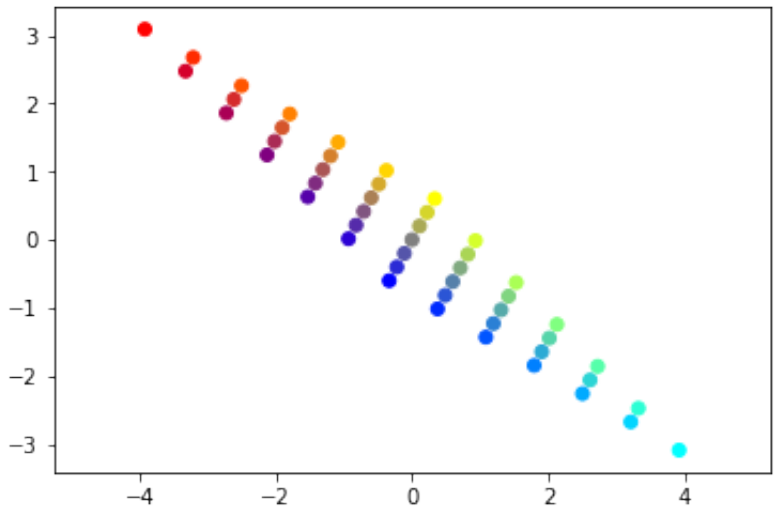
- transformation to another vector space
- movement in the space

In [33]:
```
s = 3  # grid size
x = np.arange(-s, s+1)
X1, X2 = np.meshgrid(x, x)
X = np.vstack((np.ravel(X1), np.ravel(X2)))
C = (np.vstack((X[0,:], X[1,:], -X[0,:])).T + s)/(2*s)  # RGB
plt.scatter(X[0,:], X[1,:], c=C)
p = plt.axis('square')
```



In [34]:
```
a = 1
A = np.random.random((2, 2))*2*a - a   # uniform in [-a, a]
print(A)
AX = A @ X
plt.scatter(AX[0,:], AX[1,:], c=C)
p = plt.axis('equal')
```

```
[[-0.59786955  0.7091314 ]
 [ 0.6151479  -0.41463623]]
```



In [ ]:

Determinant and inverse

The transformed space are expanded, shrunk, or flattened.
The *determinant* of a square matrix measures the expansion of the volume.
For a 2 by 2 matrix $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$,
the determinant is computed by

$$\det A = ad - bc.$$

You can use `linalg.det()` for any matrix.

In [35]:
```
A = np.array([[1, 2], [3, 4]])
np.linalg.det(A)
```

Out[35]:
```
-2.0000000000000004
```

If $\det A \neq 0$, a matrix is called *regular*, *non-singular*, or *invertible*.
The inverse A^{-1} of a square matrix A is defined as a matrix satisfying

$$AX = XA = I$$

where I is the identity matrix.

In [36]:
```
Ai = np.linalg.inv(A)
print(Ai)
print( A @ Ai, Ai @ A)
```

```
[[-2.   1. ]
 [ 1.5 -0.5]]
[[  1.00000000e+00   1.11022302e-16]
 [  0.00000000e+00   1.00000000e+00]] [[  1.00000000e+00   4.4408921
0e-16]
 [  0.00000000e+00   1.00000000e+00]]
```

In [ ]:
```
```

Solving linear algebraic equations

Many problems are formed as a set of linear equations:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m \end{aligned}$$

This can be expressed by a matrix-vector equation

$$A\boldsymbol{x} = \boldsymbol{b}$$

where

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}, \quad \boldsymbol{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}.$$

If $m = n$ and A is regular, the solution is given by

$$\boldsymbol{x} = A^{-1}\boldsymbol{b}.$$

In [37]:
```
A = np.array([[2, 2], [3, 4]])
b = np.array([1, 0])
x = np.linalg.inv(A) @ b
print('x = ', x)
print('Ax = ', A @ x)
```

```
x =  [ 2.  -1.5]
Ax =  [ 1.  0.]
```

If A^{-1} is used just once, it is more efficient to use a linear euqation solver function.

In [38]:
```
x = np.linalg.solve(A, b)
print(x)
```

```
[ 2.  -1.5]
```

In [ ]:
```
```

Pseudo inverse

When $m > n$, there is usually no solution.
When $m < n$ or A is singular, there can be many solutions.
A *pseudo* inverse A^+ is defined as a matrix satisfying $AXA = A$ (with AX and XA symmetric).
For $m > n$, $\boldsymbol{x} = A^+\boldsymbol{b}$ minimizes $||A\boldsymbol{x} - \boldsymbol{b}||$.
For $m < n$ or singular A, $\boldsymbol{x} = A^+\boldsymbol{b}$ gives a solution with minimal norm $||\boldsymbol{x}||$.

In [39]:
```
A = np.array([[1, 2, 3], [4, 5, 6]])
Api = np.linalg.pinv(A)
print( Api, A@Api)
B = A.T
Bpi = np.linalg.pinv(B)
print( Bpi, Bpi@B)
```

```
[[-0.94444444  0.44444444]
 [-0.11111111  0.11111111]
 [ 0.72222222 -0.22222222]] [[  1.00000000e+00  -2.22044605e-16]
 [ -1.77635684e-15   1.00000000e+00]]
[[-0.94444444 -0.11111111  0.72222222]
 [ 0.44444444  0.11111111 -0.22222222]] [[  1.00000000e+00  -1.77635
684e-15]
 [ -2.22044605e-16   1.00000000e+00]]
```

In [ ]:

Eigenvalues and eigenvectors

With a transformation by a symmetrix matrix A, some vectors keep its own direction. Such a vector is called an *eigenvector* and its scaling coefficient is called the *eigenvalue*.
Eigenvalues and eigenvectors are derived by solving the equation

$$A\boldsymbol{x} = \lambda\boldsymbol{x}$$

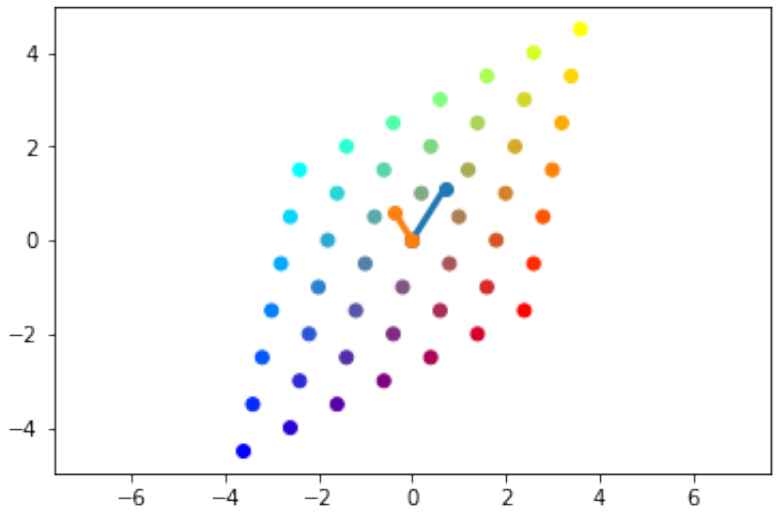
which is equivalent to $A\boldsymbol{x} - \lambda\boldsymbol{x} = (A - \lambda I)\boldsymbol{x} = 0.$
Eigenvalues are derived by solving a polynomial equation

$$\det(A - \lambda I) = 0$$

You can use `linalg.eig()` function to numerically derive eigenvalues and eigenvectors of any matrix.

In [40]:
```
A = np.array([[1, 0.2], [0.5, 1]])
lam, V = np.linalg.eig(A)
print( lam, V)
for i in range(2):
    plt.plot( [0, lam[i]*V[0,i]], [0, lam[i]*V[1,i]], 'o-', lw=3)
# colorful grid from above
AX = A @ X
plt.scatter(AX[0,:], AX[1,:], c=C)
p = plt.axis('equal')
```

```
[ 1.31622777  0.68377223] [[ 0.53452248 -0.53452248]
 [ 0.84515425  0.84515425]]
```



In [ ]:

Eigendecomposition

For a square matrix A, consider a matrix consisting of its eigenvalues on the diagonal

$$\Lambda = \mathrm{diag}(\lambda_1, \cdots, \lambda_n)$$

and another matrix made of their eigenvectors in columns

$$V = (\boldsymbol{v}_1, \cdots, \boldsymbol{v}_n).$$

From

$$AV = (\lambda_1 \boldsymbol{v}_1, \cdots, \lambda_n \boldsymbol{v}_n) = (\boldsymbol{v}_1, \cdots, \boldsymbol{v}_n) \begin{pmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{pmatrix} = V\Lambda,$$

if V is invertible, A can be represented as

$$A = V\Lambda V^{-1}.$$

This representation of a matrix is called *eigendecomposition* or *spectral decomposition*.

This representation is extremely useful in multiplying A many times as

$$A^k = V\Lambda^k V^{-1} = V\mathrm{diag}(\lambda_1^k, \cdots, \lambda_n^k)V^{-1}$$

requires only exponentials in the diagonal terms rather than repeated matrix multiplications.

It also helps intutitive understanding of how a point $\boldsymbol{x}$ transofrmed by A many times as $A\boldsymbol{x}, A^2\boldsymbol{x}, A^3\boldsymbol{x}, \ldots$ would move.

Two dimensional system

Let us take the simplest and the most important case of 2 by 2 matrix

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}.$$

We can analytically derive the eivenvalues from

$$\det(A - \lambda I) = (a - \lambda)(d - \lambda) - bc = 0$$

as

$$\lambda = \frac{a + d}{2} \pm \sqrt{\frac{(a - d)^2}{4} + bc}.$$

The correctponding eigenvectors are not unique, but given by, for example,

$$\boldsymbol{x} = \begin{pmatrix} b \\ \lambda - a \end{pmatrix}.$$

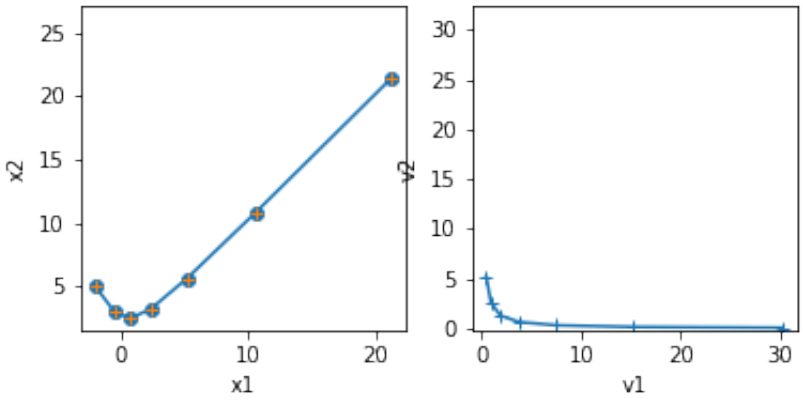
Real eigenvalues

When $\frac{(a-d)^2}{4} + bc \geq 0$, A has two real eigenvalues $\{\lambda_1, \lambda_2\}$ with corresponding eigenvectors $\{\boldsymbol{v}_1, \boldsymbol{v}_2\}$

The movement of a point $\boldsymbol{x}$ by A as $A\boldsymbol{x}, A^2\boldsymbol{x}, A^3\boldsymbol{x}, \ldots$ is composed of movements in the directions of the eigenvectors $\{\boldsymbol{v}_1, \boldsymbol{v}_2\}$. It is convergent if $|\lambda_i| < 1$ and divergent if $|\lambda_i| > 1$.

In [41]:
```
# take a 2x2 matrix
A = np.array([[1.5, 0.5], [1, 1]])
# check the eigenvalues and eigenvectors
L, V = np.linalg.eig(A)
print("L =", L)
print("V = ", V)
# take a point and see how it moves
K = 7  # steps
x = np.zeros((2, K))
x[:,0] = [-2, 5]
for k in range(K-1):
    x[:,k+1] = A @ x[:,k]  # x_{k+1} = A x_k
# plot the trajectory
plt.subplot(1,2,1)
plt.plot( x[0], x[1], 'o-')
plt.axis('square'); plt.xlabel("x1"); plt.ylabel("x2");
# In the eigenspace
y = np.zeros((2, K))
y[:,0] = np.linalg.inv(V) @ x[:,0] # map to eigenspace
for k in range(K-1):
    y[:,k+1] = L*y[:,k]  # z_{k+1} = L z_k
plt.subplot(1,2,2)
plt.plot( y[0], y[1], '+-')
plt.axis('square'); plt.xlabel("v1"); plt.ylabel("v2");
# Conver back to the original space
plt.subplot(1,2,1)
xv = (V @ y).real
plt.plot( xv[0], xv[1], '+');
```

```
L = [ 2.   0.5]
V =  [[ 0.70710678 -0.4472136 ]
 [ 0.70710678  0.89442719]]
```



In [ ]:

Complex eigenvalues

When $\frac{(a-d)^2}{4} + bc < 0$, A has a pair of complex eigenvalues

$$\lambda_1 = \alpha + i\beta \text{ and } \lambda_2 = \alpha - i\beta$$

where

$$\alpha = \frac{a+d}{2} \text{ and } \beta^2 = -\frac{(a-d)^2}{4} - bc$$

with corresponding eigenvectors

$$V = (\boldsymbol{v}_1, \boldsymbol{v}_2) = (\boldsymbol{u} + i\boldsymbol{w}, \boldsymbol{u} - i\boldsymbol{w}).$$

By eigendecomposition $A = V\Lambda V^{-1}$, a point $\boldsymbol{x}$ is converted to points in a complex plane and multipled by a complex eigenvalue, which means rotation and scaling. Points in the complex plane are then converted back in a real vector spce by multiplication with V.

To see the rotation and scaling more explicitly, we can represent $\Lambda = \begin{pmatrix} \alpha + i\beta & 0 \\ 0 & \alpha - i\beta \end{pmatrix}$ as

$$\Lambda = URU^{-1}$$

where R is

$$R = \begin{pmatrix} \alpha & -\beta \\ \beta & \alpha \end{pmatrix} = \begin{pmatrix} r\cos\theta & -r\sin\theta \\ r\sin\theta & r\cos\theta \end{pmatrix}.$$

Here $r = |\lambda| = \sqrt{\alpha^2 + \beta^2}$ is the scaling factor and θ is the angle of rotation.

We can choose U as

$$U = \frac{1}{2}\begin{pmatrix} 1 & i \\ 1 & -i \end{pmatrix}$$

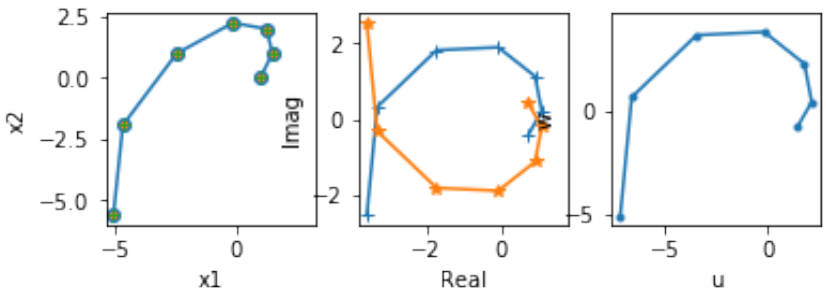
such that $VU = (\boldsymbol{u}, -\boldsymbol{w})$. Then we have another decomposition of A as

$$A = V\Lambda V^{-1} = VURU^{-1}V^{-1} = (\boldsymbol{u}, -\boldsymbol{w}) \begin{pmatrix} r\cos\theta & -r\sin\theta \\ r\sin\theta & r\cos\theta \end{pmatrix} (\boldsymbol{u}, -\boldsymbol{w})^{-1}$$

In [42]:
```
# take a 2x2 matrix
A = np.array([[1.5, -1], [1, 0.5]])
# check the eigenvalues and eigenvectors
L, V = np.linalg.eig(A)
print("L =", L)
print("V = ", V)
# take a point and see how it moves
K = 7  # steps
x = np.zeros((2, K))
x[:,0] = [1, 0]
for k in range(K-1):
    x[:,k+1] = A @ x[:,k]  # x_{k+1} = A x_k
# plot the trajectory
plt.subplot(1,3,1)
plt.plot( x[0], x[1], 'o-')
plt.axis('square'); plt.xlabel("x1"); plt.ylabel("x2");
# In the eigenspace
z = np.zeros((2, K), dtype=complex)
z[:,0] = np.linalg.inv(V) @ x[:,0]
for k in range(K-1):
```

```
    z[:,k+1] = L*z[:,k]  # z_{k+1} = L z_k
plt.subplot(1,3,2)
plt.plot( z[0].real, z[0].imag, '+-')
plt.plot( z[1].real, z[1].imag, '*-')
plt.axis('square'); plt.xlabel("Real"); plt.ylabel("Imag");
# In the cos-sin space
VU = np.c_[V[:,0].real, -V[:,0].imag]
R = np.array([[L[0].real, -L[0].imag], [L[0].imag, L[0].real]])
print("R =", R)
print("VU =", VU)
y = np.zeros((2, K))
y[:,0] = np.linalg.inv(VU) @ x[:,0]
for k in range(K-1):
    y[:,k+1] = R @ y[:,k]  # y_{k+1} = R y_k
plt.subplot(1,3,3)
plt.plot( y[0], y[1], '.-')
plt.axis('square'); plt.xlabel("u"); plt.ylabel("w");
# Conver back to the original space
plt.subplot(1,3,1)
xc = (V @ z).real
xr = VU @ y
plt.plot( xr[0], xr[1], '.')
plt.plot( xc[0], xc[1], '+');
```

```
L = [ 1.+0.8660254j  1.-0.8660254j]
V =  [[ 0.70710678+0.j          0.70710678-0.j        ]
 [ 0.35355339-0.61237244j  0.35355339+0.61237244j]]
R = [[ 1.        -0.8660254]
 [ 0.8660254  1.       ]]
VU = [[ 0.70710678 -0.        ]
 [ 0.35355339  0.61237244]]
```



In [ ]:

Covariance matrix

For m samples of n-dimensional variable $x = (x_1, \ldots, x_n)$ we usuall create a data matrix

$$X = \begin{pmatrix} x_1^1 & \cdots & x_n^1 \\ \vdots & & \vdots \\ x_1^m & \cdots & x_n^m \end{pmatrix}.$$

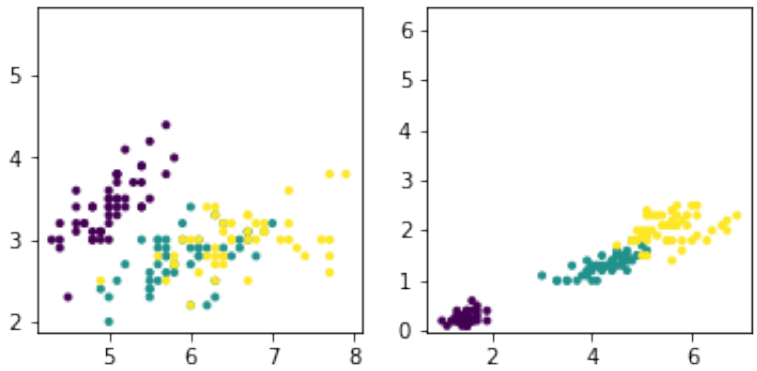
The mean vector and covariance matrix is given by

$$\bar{x} = \frac{1}{m} \sum_{j=1}^{m} x^j$$

and

$$C = \frac{1}{m-1} \sum_{j=1}^{m} (x^j - \bar{x})^T (x^j - \bar{x})$$

In [43]:
```
# Read in the iris data
X = np.loadtxt('iris.txt', delimiter=',')
Y = X[:,-1]  # flower type
X = X[:,:4]
m, n = X.shape
plt.subplot(1, 2, 1)
plt.scatter(X[:,0], X[:,1], c=Y, marker='.')
plt.axis('square')
plt.subplot(1, 2, 2)
plt.scatter(X[:,2], X[:,3], c=Y, marker='.')
plt.axis('square');
```



In [44]:
```
xbar = np.mean(X, axis=0)
dX = X - xbar  # deviation from the mean
C = (dX.T @ dX)/(m-1)
print(C)
np.cov(X, rowvar=False)
```

```
[[ 0.68569351 -0.03926846  1.27368233  0.5169038 ]
 [-0.03926846  0.18800403 -0.32171275 -0.11798121]
 [ 1.27368233 -0.32171275  3.11317942  1.29638747]
 [ 0.5169038  -0.11798121  1.29638747  0.58241432]]
```

Out[44]:
```
array([[ 0.68569351, -0.03926846,  1.27368233,  0.5169038 ],
       [-0.03926846,  0.18800403, -0.32171275, -0.11798121],
       [ 1.27368233, -0.32171275,  3.11317942,  1.29638747],
       [ 0.5169038 , -0.11798121,  1.29638747,  0.58241432]])
```

In [ ]:
```
```

Principal component analysis (PCA)

In taking a grasp of high-dimensional data, it is often useful to project the data onto a subspace where the data vary most.

To do that, we first take the covariance matrix C of the data, compute its eigenvalues and eigenvectors, and project the data onto the subspace spanned by the eigenvectors with largest eigen values. The eigenvectors of the covariance matrix is called *principal components*, ordered by the magnitude of their eigenvalues.

Because the covariance matrix is symmetric, its eigendecompotision is given by

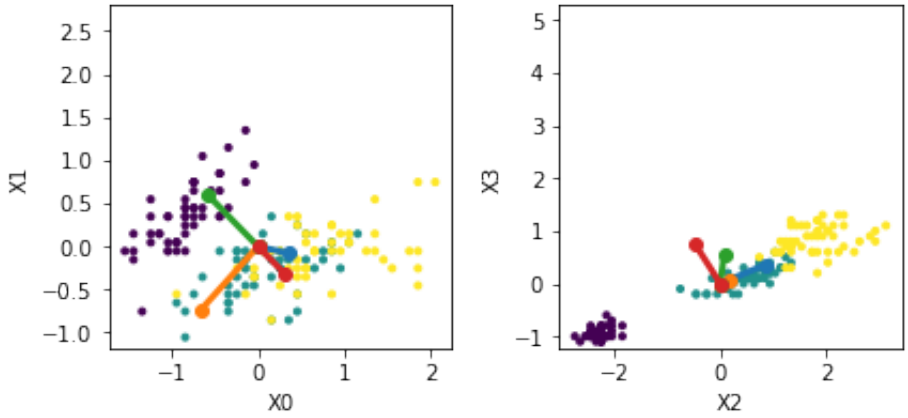
$$C = V\Lambda V^T.$$

In [45]:
```
lam, V = np.linalg.eig(C)
print('lam, V = ', lam, V)
# it is not guaranteed that the eigenvalues are sorted, so sort them
ind = np.argsort(-lam)  # indices for sorting, descending order
lams = lam[ind]
Vs = V[:,ind]
print('lams, Vs = ', lams, Vs)
```

```
lam, V =  [ 4.22484077  0.24224357  0.07852391  0.02368303] [[ 0.361
58968 -0.65653988 -0.58099728  0.31725455]
 [-0.08226889 -0.72971237  0.59641809 -0.32409435]
 [ 0.85657211  0.1757674   0.07252408 -0.47971899]
 [ 0.35884393  0.07470647  0.54906091  0.75112056]]
lams, Vs =  [ 4.22484077  0.24224357  0.07852391  0.02368303] [[ 0.3
6158968 -0.65653988 -0.58099728  0.31725455]
 [-0.08226889 -0.72971237  0.59641809 -0.32409435]
 [ 0.85657211  0.1757674   0.07252408 -0.47971899]
 [ 0.35884393  0.07470647  0.54906091  0.75112056]]
```

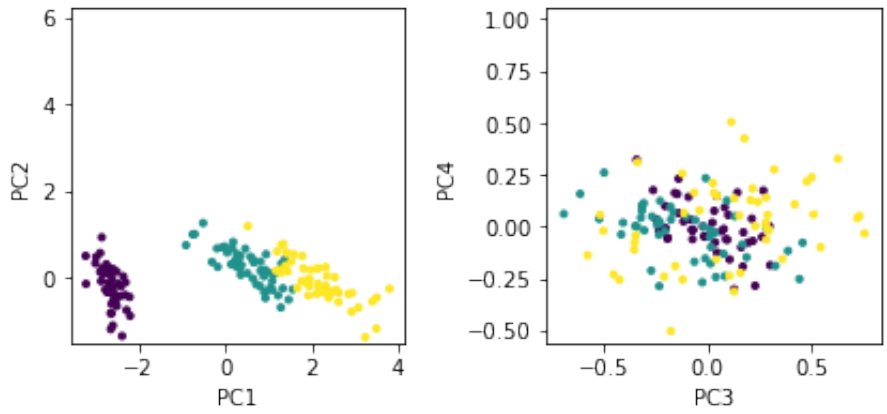
Let us see the principal components in the original space.

In [46]:
```
zero = np.zeros(n)
plt.subplot(1, 2, 1)
plt.scatter(dX[:,0], dX[:,1], c=Y, marker='.')
for i in range(4):
    plt.plot( [0, Vs[0,i]], [0, Vs[1,i]], 'o-', lw=3)
plt.setp(plt.gca(), xlabel='X0', ylabel='X1')
plt.axis('square')
plt.subplot(1, 2, 2)
plt.scatter(dX[:,2], dX[:,3], c=Y, marker='.')
for i in range(4):
    plt.plot( [0, Vs[2,i]], [0, Vs[3,i]], 'o-', lw=3)
plt.setp(plt.gca(), xlabel='X2', ylabel='X3')
plt.axis('square')
plt.tight_layout()  # adjust space
```



Now let us project the 4D data onto the space spanned by the eigenvectors.

In [47]:
```
Z = dX @ Vs
plt.subplot(1, 2, 1)
plt.scatter(Z[:,0], Z[:,1], c=Y, marker='.')
plt.setp(plt.gca(), xlabel='PC1', ylabel='PC2')
plt.axis('square')
plt.subplot(1, 2, 2)
plt.scatter(Z[:,2], Z[:,3], c=Y, marker='.')
plt.setp(plt.gca(), xlabel='PC3', ylabel='PC4')
plt.axis('square')
plt.tight_layout()  # adjust space
```



In [ ]:

Singular value decomposition (SVD)

For a $(m \times n)$ matrix A, a non-negative value $\sigma > 0$ satisfying

$$Av = \sigma u$$
$$A^T u = \sigma v$$

for unit vectors $||u|| = ||v|| = 1$ is called the *singular value*.
u and v are called left- and right-singular vectors.

Singular value decomposition (SVD) of a $(m \times n)$ matrix A is

$$A = USV^T$$

where $S = \mathrm{diag}(\sigma_1, \ldots, \sigma_k)$ is a diagonal matrix made of $k = \min(m, n)$ singular values, U is an $(m \times k)$ orthogonal matrix ($U^T U = I$) made of columns of left-singular vectors, and V is an $(n \times k)$ orthogonal matrix ($V^T V = I$) made of columns of right-singular vectors.

SVD represents a matrix by the sum of products of row and column vectors, such as spatial patterns scaled by different time courses.

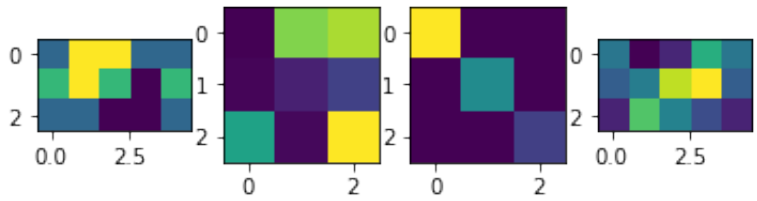
In [48]:
```
A = np.array([[0,2,2,0,0], [1,2,1,-1,1], [0,0,-1,-1,0]])
U, S, Vt = np.linalg.svd(A, full_matrices=False)
#V = Vt.T
print(U, S, Vt)
U @ np.diag(S) @ Vt
```

```
[[-0.71180091  0.45041064  0.53895243]
 [-0.69256656 -0.57789988 -0.43172132]
 [ 0.11700867 -0.68056006  0.723289  ]] [ 3.76385911  1.82308721  0.
7139451 ] [[-0.18400438 -0.74623806 -0.5933211   0.15291696 -0.18400
438]
 [-0.3169897  -0.13986082  0.55042976  0.69029058 -0.3169897 ]
 [-0.6046982   0.30039036 -0.10799914 -0.4083895  -0.6046982 ]]
```

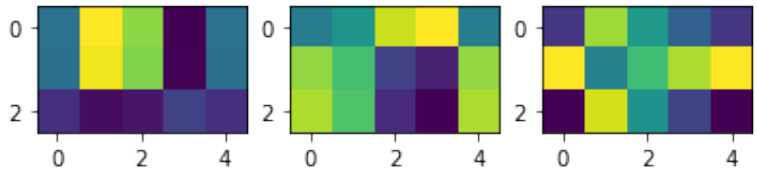
Out[48]:
```
array([[ -5.21362502e-17,   2.00000000e+00,   2.00000000e+00,
          4.38908397e-16,  -5.64805298e-17],
       [  1.00000000e+00,   2.00000000e+00,   1.00000000e+00,
         -1.00000000e+00,   1.00000000e+00],
       [ -1.34177813e-16,   2.15343687e-16,  -1.00000000e+00,
         -1.00000000e+00,  -3.63500435e-16]])
```

In [49]:
```
plt.subplot(1,4,1)
plt.imshow(A)
plt.subplot(1,4,2)
plt.imshow(U)
plt.subplot(1,4,3)
plt.imshow(np.diag(S))
plt.subplot(1,4,4)
plt.imshow(Vt)
```

Out[49]:
```
<matplotlib.image.AxesImage at 0x11296a9b0>
```



In [50]:
```
k = 3
for i in range(k):
    plt.subplot(1,k,i+1)
    plt.imshow(np.outer(U[:,i], Vt[i,:]))
```



In [ ]:
```
```

PCA by SVD

From $X = USV^T$, we can see $C = \frac{1}{m-1}X^TX = \frac{1}{m-1}VS^2V^T$.
Thus columns of V are principal components and $\sigma_j^2/(m-1)$ are their eigenvalues.

In [51]:
```
# from iris data
print(lams, Vs)  # computed by eig of covariance matrix
U, S, Vt = np.linalg.svd(dX, full_matrices=False)
print(S**2/(m-1), Vt.T)
```

```
[ 4.22484077  0.24224357  0.07852391  0.02368303] [[ 0.36158968 -0.6
5653988 -0.58099728  0.31725455]
 [-0.08226889 -0.72971237  0.59641809 -0.32409435]
 [ 0.85657211  0.1757674   0.07252408 -0.47971899]
 [ 0.35884393  0.07470647  0.54906091  0.75112056]]
[ 4.22484077  0.24224357  0.07852391  0.02368303] [[ 0.36158968 -0.6
5653988  0.58099728  0.31725455]
 [-0.08226889 -0.72971237 -0.59641809 -0.32409435]
 [ 0.85657211  0.1757674  -0.07252408 -0.47971899]
 [ 0.35884393  0.07470647 -0.54906091  0.75112056]]
```

In [ ]:

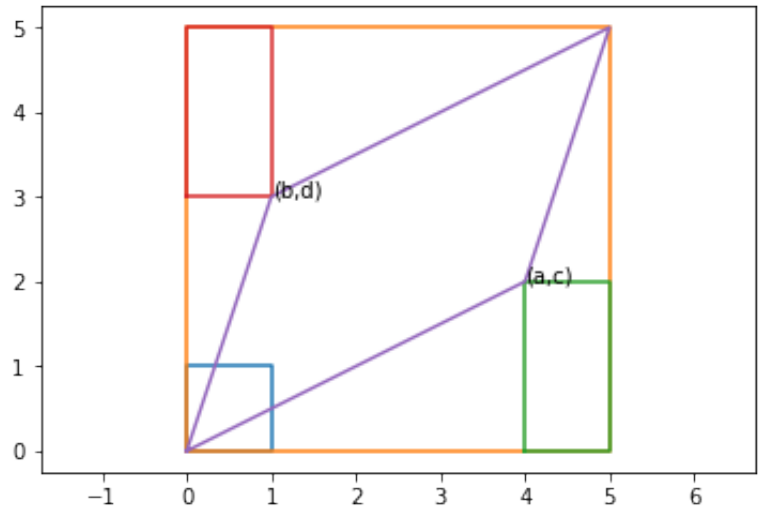
Exercise

1. Determinant and eigenvalues

1) For a 2x2 matrix $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$, verify that $\det A = ad - bc$ in the case graphically shown below (a, b, c, d are positive).

In [52]:
```
A = np.array([[4, 1], [2, 3]])
plt.plot([0, 1, 1, 0, 0], [0, 0, 1, 1, 0])
plt.plot([0, A[0,0]+A[0,1], A[0,0]+A[0,1], 0, 0],
         [0, 0, A[1,0]+A[1,1], A[1,0]+A[1,1], 0])
plt.plot([A[0,0], A[0,0]+A[0,1], A[0,0]+A[0,1], A[0,0], A[0,0]],
         [0, 0, A[1,0], A[1,0], 0])
plt.plot([0, A[0,1], A[0,1], 0, 0],
         [A[1,1], A[1,1], A[1,0]+A[1,1], A[1,0]+A[1,1], A[1,1]])
plt.plot([0, A[0,0], A[0,0]+A[0,1], A[0,1], 0],
         [0, A[1,0], A[1,0]+A[1,1], A[1,1], 0])
plt.axis('equal')
plt.text(A[0,0], A[1,0], '(a,c)')
plt.text(A[0,1], A[1,1], '(b,d)')
```

Out[52]:
```
<matplotlib.text.Text at 0x111d95be0>
```



A unit square is transformed into a parallelogram. Its area S can be derived as follows:
Large rectangle: $S_1 = (a + c)(b + d)$
Small rectangle: $S_2 =$
Bottom/top triangle: $S_3 =$
Left/right triangle: $S_4 =$
Parallelogram: $S = S_1 - \ldots$

2) The determinant equals the product of all eigenvalues. Verify this numerically for multiple cases and explain intuitively what that should be the case.

In [ ]:
```
A = np.array(...
det =
print('detA = ', det)
lam, V =
print(np.product(lam))
```

The determinant represents ...

The eigenvalues mean ...

Therefore, ...

In [ ]: