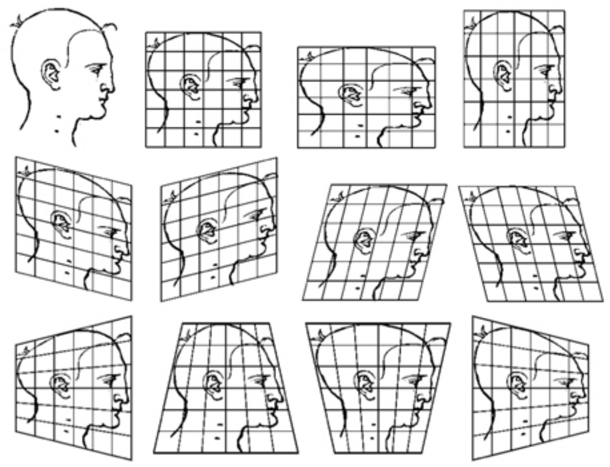
Python #9
Linear Algebra

https://people.sc.fsu.edu/∼jburkardt/classes/python_2022/python09/python09.pdf
Freely adapted from the Python lessons at https://software-carpentry.org/



Linear Algebra

- *Mathematicians think of lists and tables as vectors and matrices;*
- *Vectors are thought of as "things"*
- *Matrices represent linear transformations of vectors;*
- *The study of linear transformations is called linear algebra;*
- *Python's* `numpy` *library gives us tools for linear algebra;*
- *Vectors have norm (length), unit direction, pairwise angle;*
- *Matrix-vector multiplication A*x=b transforms x into b;*
- *Given A and b, we can usually figure out what x was;*
- *Insight into a matrix comes from LU, QR, SVD factorizations;*
- *Matrix eigenvalues indicate the structure of the transformation.*

Machine learning looks for patterns and structures in data. Linear relationships are often the best simple approximation to complicated systems. Linear algebra is a collection of ideas and tools that we can use to construct these simple models of our observations.

In linear algebra, we study abstract objects called *vectors*; in machine learning, these are the individual observations of temperature, answers on a survey, medical records. While in machine learning, we might have a table whose rows are the observations, linear algebra thinks of this as a *matrix.* Linear algebra can help us decide whether

- two observations are very similar;
- your hospital bill can be approximately predicted by your sex, age, and smoking status;
- an image we have scanned is a picture of a cat or a dog.

The Linear Algebra Problem

How can we use the tools of linear algebra to analyze our data when we think of it as vectors? In particular, we want to:

- *distinguish lists, row vectors, and column vectors;*
- *initialize a vector;*
- *compute the norm (size) of a vector;*
- *compute the distance and angle between two vectors;*
- *initialize a matrix;*
- *multiply a vector by a matrix;*
- *solve a system of linear equations;*

1 Lists and arrays

A vector is the fundamental object of linear algebra, which studies the ways in which a linear transformation, which might be represented by A, converts a vector x into a vector y. This process is often written in the form $A * x = y$.

We tend to think of a vector as simply a list of numbers. In Python, we have already seen that lists of numbers can be created using square brackets and commas:

```
  my_list = [ 1, 2, 3 ]
```

However, especially for numerical work, we will prefer to use a library called `numpy`, which allows us to do many numerical operations that are not defined for the standard Python list. When we're doing basic vector operations, we can then get away with creating an array by importing `numpy`, and then creating the array as a list of values, input to `np.array()`:

```
  import numpy as np
  x = np.array ( [ 1, 2, 3 ] )
```

There are several facts we can check for our array:

```
  type ( x )
    <class 'numpy.ndarray'>
  x.shape
    (3,)        # The trailing comma here has no special meaning.  x is a list of 3 things.
  len ( x )
    3
```

You may be aware that mathematics (and MATLAB!) distinguish between row and column vectors. However, our arrays are neither row or column vectors. They hold the number information, but no shape information. As long as possible, we will prefer to work with the arrays that Python sets up so easily. We will find that we can often get away with ignoring the distinction between row and column vectors. But, in case we need to create or work with such objects, we will look at this issue in the next section.

Just be aware that the simple array definition means that some common linear algebra operations won't work the way we expect. For instance, in mathematics (and MATLAB!), there is a *transpose* operation, that converts a row vector to a column vector, or vice versa. There is a corresponding transpose operation in Python (which works just fine for matrices!) but what it does to our simple array might surprise you.

```
x
  array([1, 2, 3])
x.T                   #  Adding ".T" will transpose an object
```

```
  array([1, 2, 3])
np.transpose(x)      # ...or calling np.transpose()
  array([1, 2, 3])
```

2 Row and column vectors

The situation is more complicated if we wish to follow mathematical rules while working with vectors. That's because, in linear algebra, we distinguish between *row vectors* and *column vectors*. We describe the shape of a vector as $m \times n$, where m is the number of rows, and n is the number of columns. For vectors, at least one of these two numbers must be 1. (Otherwise, we would be describing some matrix, which we are not ready to think about yet!) For example, a 3×1 row vector r might look something like this:

$$r = \begin{pmatrix} 10 & 12 & 7 \end{pmatrix}$$

while a 1×4 column vector c might be written this way:

$$c = \begin{pmatrix} 14 \\ 92 \\ 17 \\ 76 \end{pmatrix}$$

The problem we saw earlier, when trying to transpose our array `x` is that it is not really a mathematical row or column vector. If we really need to work with row or column vectors, we have create an object that looks like a list of lists. That is, we create a list, and each item in the list is the contents of a row of the object.

Here's how we would set up our r and c arrays, and examine the results:

```
r = np.array ( [ [ 1, 2, 3 ] ] )
r.shape
  (1, 3)
len(r)
  1
r.T
  array([[1],
         [2],
         [3]])

c = np.array ( [ [ 1 ], [ 2 ], [ 3 ] ] )
c.shape
  (3, 1)
len (c )
  3
c.T
  array([[1, 2, 3]])
```

3 One array or vector

We can think of a vector as a one-dimensional list of values. The number of values is the *dimension* or *extent* of the vector. We often refer to this extent as `n`. If v has extent n, we may express this by "v is an n-vector".

A vector has a geometric interpretation. We can draw a two-dimensional vector $[v_1, v_2]$, for instance, on a Cartesian grid, as an arrow starting at the origin and extending to the point with coordinates (v_1, v_2). Thus, we say that, geometrically, a vector has a length and a direction.

To measure the *length* or *magnitude* or *norm* of a vector, we combine its values in a certain way. Although there are many vector norms, the most common and useful are the *Euclidean* or *"2"* norm::

$$||v||_2 = \sqrt{\sum_{i=1}^{n} v_i^2}$$

and the maximum norm or *"infinity"* norm:

$$||v||_\infty = \max_{0 \leq i < n} |v_i|$$

In Python, the Euclidean norm of vector v can be computed by `np.linalg.norm(v)` and the max norm by `np.linalg.norm(v,np.inf)`:

```
import numpy as np
x = np.random.rand ( 2 )
print ( '2Norm of', x, ' is ', np.linalg.norm ( x ) )
print ( 'MNorm of', x, ' is ', np.linalg.norm ( x, np.inf ) )
y = np.array ( [ 3, 4 ] )
print ( '2Norm of', y, ' is ', np.linalg.norm ( y ) )
print ( 'MNorm of', y, ' is ', np.linalg.norm ( y, np.inf ) )
z = np.ones ( 2 )
print ( '2Norm of', z, ' is ', np.linalg.norm ( z ) )
print ( 'MNorm of', z, ' is ', np.linalg.norm ( z, np.inf ) )
```

From now on, if we simply say *norm*, we will mean the Euclidean norm.

The norm computes the length of a vector. If we divide a vector v by its Euclidean norm, we get what is called a *unit vector*, sometimes written $\hat{v}$.

$$\hat{v} = \frac{v}{||v||}$$

For instance, the vector $v = [2, 3]$ has a norm of $\sqrt{13}$, and so $\hat{v} = [\frac{2}{\sqrt{13}}, \frac{3}{\sqrt{13}}]$. You can verify that $||\hat{v}|| = 1$.

We can think of $\hat{v}$ as the *direction* of the vector, while $||v||$ is the *strength* or *magnitude*. Thus, simply rewriting the definition of the unit vector, we can factor any (nonzero) vector into a direction and magnitude:

$$v = ||v||\,\hat{v}$$

Note that if we multiply a vector v by a numeric factor α to get a vector w, this changes the norm of the vector, but not the direction.

$$w = \alpha * v$$
$$||w|| = \alpha * ||v||$$
$$\hat{w} = \frac{w}{||w||}$$
$$\hat{v} = \frac{v}{||v||}$$
$$\hat{w} = \hat{v}$$

You can verify this by considering $w = 10 * v = 10 * [2, 3] = [20, 30]$, computing $\hat{w}$, and showing that it equals $\hat{v}$.

4 The case of two vectors

If we have two vectors v and w of the same extent n, there are some obvious questions we can ask:

- Can we add w and v?
- Is w equal to v?
- Is w simply a multiple of v (longer? shorter?)
- Does w lie somewhat or very little, along the direction of v?
- What is the angle between v and w?

Geometrically, we can think of adding two vectors as a kind of two-stage walk. We start at the origin, then walk in the direction and length of vector w. Once we reach the end of that first stage, we continue to walk from there, but now in the direction and length of v. You can make a corresponding plot by drawing the vector w starting at the origin, and then adding a picture of v that starts at the tip of w.

Algebraically, things are very simple. The sum of w and v is a new vector we can call u, whose components are found by adding the corresponding pairs of components of w and v:

$$u_i = w_i + v_i$$

It turns out that w is equal to v exactly if the norm of $v-w$ is zero. So this is easy to check! In fact, $||v-w||$ measures the magnitude or distance between the two vectors, so the size of this quantity is an indication of how close they are.

To answer the other questions, we need to know about the vector *inner product*, usually known by the more familiar name of *dot product*:

$$< v, w >= \sum_{i=1}^{n} v(i) * w(i)$$

It turns out that

$$< v, w >= ||v|| \cdot ||w|| \cdot \cos(\alpha)$$

where α is the angle between the two vectors. In particular, this means that

$$\cos(\alpha) = \frac{< v, w >}{||v|| \, ||w||}$$

If w is simply a multiple of v, then

- $\cos(\alpha) = 1$ (the vectors point in the same direction), or
- $\cos(\alpha) = -1$ (the vectors point in opposite directions.)

On the other hand, if $\cos(\alpha) = 0$, then the vectors are perpendicular.

In general, two vectors will have $\cos(\alpha)$ somewhere between -1 and +1. Values near +1 or -1 mean the two directions are close, while values near 0 mean they have little relation. In our work, we will often need to use this measurement to decide whether two objects, represented by vectors, are closely related or not.

In Python, we can compute the vector dot product of v and w using `np.dot(v,w)`

```
import numpy as np
w = np.random.rand ( 2 )
x = np.random.rand ( 2 )
wdotx = np.dot ( w, x )
print ( 'dot(w,x) = ', wdotx )
```

and the cosine of α is also easy:

```
import numpy as np
w = np.random.rand ( 2 )
w_norm = np.linalg.norm ( w )
x = np.random.rand ( 2 )
x_norm = np.linalg.norm ( x )
cwx = np.dot ( w, x ) / w_norm / x_norm
print ( 'cos(w,x) = ', cwx )
```

Notice in these examples that w and x are defined as simple numpy arrays (essentially lists). We did not try to define them as row or column vectors.

In mathematics, and in MATLAB, the vector dot product can only be applied to a row vector dotted with a column vector. If the dot product of two column vectors, say c_1 and c_2 is desired, then the first factor must be transposed before the dot product is computed. Mathematically this might be written as

$$\text{c1dotc2} = c_1^T \cdot c_2$$

or in MATLAB, as

```
c1dotc = c1' * c2
```

If you are used to the conventions of mathematics or MATLAB, you should be aware that the `np.dot()` function, can be applied to simple `numpy` arrays as long as they have the same extent (number of entries).

5 The projection of one vector on another

Given any two vectors, it is unlikely that they are equal, but we would still like some information about whether they are closely related or not. We already know how to compute the angle between vectors u and v. However, another way to view this relationship is to measure what is called the *projection* of v onto u. What we want to know is, how much of vector v is going in the direction of u?

We are going to compute a dot product between v and the *direction* of u. In other words, we compute a number which we might call α:

$$\alpha = < v, \hat{u} >$$

Now we can think of $v = v_1 + v_2$, where the v_1 component is in the direction u, and the v_2 component is perpendicular to u. Thus we can also compute v_2:

$$\begin{aligned}
v_1 &= \alpha * \hat{u} \\
v &= v_1 + v_2 = \alpha * \hat{u} + v_2 \\
v_2 &= v - \alpha * \hat{u} \\
v &= v_1 + v_2 = \alpha * \hat{u} + (v - \alpha * \hat{u})
\end{aligned}$$

As an example of projection, let $u = (3, 4)$, and $v = (5, 1)$:

```
  u = np.array ( [ 3, 4 ] )
  u_norm = np.linalg.norm ( u )
  uhat = u / u_norm
  uhat_norm = 1.0
  v = np.array ( [ 5, 1 ] )
  v_norm = np.linalg.norm ( v )

  beta = np.dot ( v, uhat )
  v1 = beta * uhat
  v2 = v - v1
  v1_norm = np.linalg.norm ( v1 )
  v2_norm = np.linalg.norm ( v2 )

  cos1 = np.dot ( v1, uhat ) / v1_norm / uhat_norm
  angle1 = np.arccos ( cos1 )     # Python may complain cos1 illegal input for arccos
  cos2 = np.dot ( v2, uhat ) / v2_norm / uhat_norm
  angle2 = np.arccos ( cos2 )
```

You may find that the computation of `angle1` fails, because the value of `cos1` is illegal as input to `np.arccos()`. If so, look carefully at the value of `cos1`, explain what is wrong, and figure out a way to fix the problem.

Projection decomposes a vector v into two components v_1 and v_2 which are perpendicular to each other. This means that the three vectors v, v_1, and v_2 form a 90 degree triangle, and hence we must have $||v||^2 = ||v_1||^2 + ||v_2||^2$. You can verify this from the previous calculations by printing and comparing `v_norm**2` and `v1_norm**2+v2_norm**2`.

6 A matrix is a two dimensional array

In linear algebra, a matrix is a two-dimensional table of numbers, with m rows and n columns. In machine learning, a matrix has two common uses. A matrix can be used to represent m examples of data, each having n measurements. It can also be used to represent a linear relationship that we discover between some of the measurements in a set of data. But this discussion will be useful, no matter which way we happen to be using a matrix in our machine learning application.

As we have seen, one of Python's data types is the list, a set of elements surrounded by square brackets. The elements need not be numbers; they can be just about any data type. That means the elements could be lists themselves, so Python easily handles lists of lists. Just to be clear, here is how such a list might be created:

```
  menu = [ [ 'calzone', 'lasagna', 'pasta', 'pizza' ],
           [ 'burrito', 'gazpacho', 'taco' ],
           [ 'gyro', 'hummus', 'tabouli' ] ]
```

However, for numerical work, this simple list-of-lists data type is not suitable to handle matrices. To start with, we expect a matrix to contain only numeric values, and we expect the matrix to be *rectangular*, that is, to be arranged as an $m \times n$ table of values. Neither of these features are guaranteed for a Python list-of-lists. Instead, we will always use the `numpy` library to create matrices, which will guarantee that they are numeric arrays in tabular format. Given this assurance, `numpy` supplies a huge library of functions that we can use for linear algebra operations involving such matrices.

We have seen that a `numpy` array `v` can be initialized by listing its values using the `v=np.array([`*list of m values*`]))` function. A matrix, for `numpy`, is initialized in the same way as a list of lists. In particular, the matrix is defined as a list of rows, and each row is a list of numeric values. The matrix

$$A = \begin{pmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \end{pmatrix}$$

could be entered in Python by

```
import numpy as np
A = np.array ( [
   [ 11, 12, 13, 14 ],
   [ 21, 22, 23, 24 ],
   [ 31, 32, 33, 34 ] ] )
```

As we saw earlier for simple arrays, we can get some useful type, shape, and size information about a matrix:

```
type ( A )
A.shape
len ( A )
A.T
```

Other useful matrix creation commands include

```
import numpy as np
B = np.zeros ( [ 4, 5 ] )     # A 4x5 array of 0's
C = np.ones ( [ 6, 3 ] )      # a 6x3 array of 1's
D = np.random.rand ( 2, 5 )   # 2x5 random values
I = np.identity ( 3 )         # 3x3 identity matrix
```

7 $A * x = y$: A matrix can multiply a vector

Matrices affect vectors by multiplying them. To compute the matrix-vector product $A * x$, if A is an $m \times n$ matrix, then x must be an n-vector and y must be an m-vector. One way to think about this is that if you replace the equation $A * x = y$ by the shapes of the objects, you must have something like

$$(m \times n) \times (n \times 1) = (m \times 1)$$

On the left hand side, the second and third dimensions must be equal ($n = n$), otherwise the multiplication is illegal. The first and fourth dimensions define a shape $(m, 1)$ that must match the shape of the object on the right hand side.

If you are unfamiliar with the rules of matrix-vector multiplication, then you need to look online for help. Briefly, each result y_i is computed by multiplying corresponding elements of row i of matrix A with elements of x:

$$y_i = \sum_{j=1}^{n} A_{i,j}\, x_j$$

but it is my conviction that no one ever understood matrix multiplication by looking at such a formula. That's why, if this is unfamiliar to you, I urge you to find a nice explanation online that includes diagrams and examples!

We can multiply a matrix times a vector using `np.matmul()`:

```
import numpy as np
A = np.array ( [
  [ 1, 2, 3 ],
  [ 4, 5, 6 ],
  [ 7, 8, 0 ] ] )
x = np.array ( [ 1, 2, 3 ] )
y = np.matmul ( A, x )
print ( A, '*', x, '=', y )
```

In fact, we can use the same function `np.matmul(A,B)=C` to compute a matrix-matrix product as well. In order for this product to be legal, we check dimensions:

$$(m_1 \times n_1) \times (m_2 \times n_2) = (m_3 \times n_3)$$

and we can state that the multiplication can only be carried out if $n_1 = m_2$, in which case the dimensions (m_3, n_3) of the product C will actually be equal to (m_1, n_2).

You can investigate this by computing the product of two random matrices of unusual shape:

```
A = np.random.rand ( 3, 2 )
B = np.random.rand ( 2, 5 )
C = np.matmul ( A, B )
```

State why the computation of C is legal, and what the shape of C will be!

8 Matrix norms

We have seen how to compute the norm of a vector, which assigns a sort of size to it. It is also useful to define the norm of a matrix. But while a vector is a kind of thing, an arrow, which clearly has geometric length, a matrix is really an operator, something that modifies vectors. Therefore, the norm of a matrix will measure, in a sense, "how much" the matrix changes a vector. In general, the norm of a matrix A will be defined as the maximum change that it can make to any vector. We want this change to be measured relative to the original size of the vector, so our definition will be something like this:

$$||A|| = \max_{x \neq 0} \frac{||A * x||}{||x||}$$

A matrix norm defined in this way is said to be *vector bound.* The actual value we get for $||A||$ depends on our choice of vector norm on the right hand side of the definition. It is common to use $||A||_2$ to indicate that our matrix norm was defined based on the $l2$ vector norm, with similar notations $||A||_1$ and $||A||_\infty$ for the $l1$ and $l\infty$ norms. Typically, we know what norm is being used, and can omit the subscript.

The important fact to note, however, is that as long as we use the vector-bound matrix norm, we can always bound a matrix-vector product:

$$||Ax|| \leq ||A||\,||x||$$

There is one more norm, known as the Frobenius norm, denoted by $||A||_F$, which is not computed in the usual vector-bound way. However, it is much easier to compute than the $l2$ matrix norm, and it is always the case that $||A||_2 \leq ||A||_F$. That means we can use $||A||_F$ as a rough but cheaper estimate of $||A||_2$, and we can still bound a matrix-vector product, although our upper bound may be a little bigger than if we had used the $||A||_2$ norm itself.

There are methods for computing the norm directly from the matrix values:

- $||A||_1$: maximum sum of entries in any column of A;
- $||A||_2$: maximum eigenvalue of $A'A$ or maximum singular value of A;
- $||A||_\infty$: maximum sum of entries in any row of A;
- $||A||_F$: square root of sum of squares of all entries of A;

9 Matrix norms in Python

Matrix norms are available via the function `np.linalg.norm()`, which is the same function we use for vector norms. Luckily, Python can figure out which kind of object we are examining. Typical calls include:

```
  A_norm_one = np.linalg.norm ( A, 1 )
  A_norm_two = np.linalg.norm ( A, 2 )
  A_norm_inf = np.linalg.norm ( A, np.inf )
  A_norm_fro = np.linalg.norm ( A, 'fro' )
  A_norm_fro = np.linalg.norm ( A )
```

Notice that, unless you specify a second argument to choose your matrix norm, Python will use the Frobenius norm for matrices, whereas mathematicians prefer $||A||_2$.

The importance of norms comes from the necessity for measuring (and controlling) error in linear algebra computations. Often, these computations involve a step in which we multiply a vector x by a matrix A, getting a result y. If we know the sizes (norms) of x and A in advance, we may want to be able to produce a guaranteed maximum on the size of the result vector y. Norms give us this ability.

For example, if I start out with vectors x inside the circle of radius 2 (that is, $||x|| \leq 2$), and $||A|| = 10$, then I know that every product vector $y = A*x$ must lie inside the circle of radius 20, (because $||y|| \leq 10*||x|| \leq 20$ Let us verify this for our example:

```
import numpy as np
A = np.array ( [
  [ 1, 2, 3 ],
  [ 4, 5, 6 ],
  [ 7, 8, 0 ] ] )
A_norm = np.linalg.norm(A,2)
print ( '  A_norm = ', A_norm )
A_norm_estimate = 0.0
for test in range ( 0, 100 ):
  x = np.random.rand ( 3 )
  x_norm = np.linalg.norm ( x, 2 )
  y = np.matmul ( A, x )
  y_norm = np.linalg.norm ( y, 2 )
  A_norm_estimate = max ( A_norm_estimate, y_norm / x_norm )
print ( '  A_norm_estimate = ', A_norm_estimate )
```

Notice that we specified the use of the Euclidean or "2-norm" for both matrices and vectors. When doing this kind of calculation, the same norm must be used for both cases. Unfortunately, in `numpy`, the default matrix norm is Frobenius, while the default vector norm is Euclidean. So we had to be explicit in our norm choice.

As an exercise, suppose that the matrix A and vector y are defined as:

$$A = \begin{pmatrix} 5 & 7 & 6 & 5 \\ 7 & 10 & 8 & 7 \\ 6 & 8 & 10 & 9 \\ 5 & 7 & 9 & 10 \end{pmatrix}, \qquad y = \begin{pmatrix} 58 \\ 81 \\ 77 \\ 69 \end{pmatrix}$$

Write a Python program in which you:

1. set the variables `A` and `y`;
2. solve for a vector `x` so that $A * x = y$;
3. compute $||A||$, $||x||$, $||y||$ and verify that $||y|| \leq ||A|| * ||x||$
4. compute the error $e = A * x - y$, and print $||e||$.

10 Estimate the norm of a matrix

The norm of a matrix `A` can be defined in terms of a corresponding vector norm, as follows:

$$||A|| = \max_{||x|| \neq 0} (\frac{||Ax||}{||x||})$$

This suggests one way to estimate a matrix norm - simply generate a lot of random vectors `x`, compute the ratios $\frac{||Ax||}{||x||}$ and use the maximum as an estimate.

Consider the following matrix `A`:

```
A = np.array ( [
   [  0, -1,  2 ],
   [ -3,  4, -5 ],
   [  6, -7,  8 ] ] )
```

1. Use this technique to estimate the norm of `A` under each of the $l_1, l_2, l_|infty$ norms.
2. Choose one of these norms, and try 10, 100, 1000, 1000 random vectors and observe whether the estimates converge to the correct value.
3. Compare your results when you use uniform random values for the test vector `x`, or normal random values.

11 `np.linalg.solve()`: Solving $A * x = y$ for x

It turns out that many linear algebra problems seek to reverse the process of matrix multiplication. In other words, we know the matrix A and the right hand side y, and we wish to find a vector x so that $A*x = y$. This is called *solving a linear system.* For now, we will assume that the matrix A is square (that is, that $m = n$). Mathematically, we should also assume that the matrix A is not *singular.* When a matrix is singular, the solution process can break down. For now, we will just assume that's not going to happen.

As mentioned in class, Gauss elimination can be used to solve the system. Luckily for us, the function `np.linalg.solve(A,y)` will do this for us:

```
import numpy as np
A = np.array ( [
   [ 1, 2, 3 ],
   [ 4, 5, 6 ],
   [ 7, 8, 0 ] ] )
y = np.array ( [ 14, 32, 23 ] )
x = np.linalg.solve ( A, y )
print ( A, '*', x, '=', y )
#
#  Verify norm of error is zero or very small:
#
e = np.matmul ( A, x ) - y
e_norm = np.linalg.norm ( e )
print ( 'Error ||A*x-y|| = ', e_norm )
```

Although the existence of Gauss elimination would seem to settle the question of solving linear systems, we will soon see that this is not the end of the matter. We will want to make some new approaches if the matrix A is very large but mostly zero ("sparse"), or is rectangular, with too many equations, or too many variables, or has some special property that suggests a better way of solving linear systems.

12 `sp.linalg.lu()`: The PLU factorization

Gauss elimination is the standard method of solving a linear system. For numerical work, it is often useful to employ a version of the algorithm that produces a special factorization of the original matrix `A` into the product `P * L * U`, where

- `P` is a permutation matrix;
- `L` is a lower triangular matrix with unit diagonal;
- `U` is an upper triangular matrix;

Once this factorization is computed, it is easy to solve multiple linear systems or compute the determinant or inverse matrix.

In the `scipy()` sublibrary library `linalg()`, there is a function `lu()` which can compute this factorization.

```
from scipy.linalg import lu
P, L, U = lu ( A, permute_l = False )
```

13 sp.linalg.qr(): The QR factorization

The QR factorization is applied to an $m \times n$ matrix `A`, returning factors

- `Q` an $m \times m$ orthogonal matrix
- `R` an $m \times n$ upper triangular matrix;

so that $A = Q * R$.

If we are considering a linear system $A * x = b$, then the solution is $x = R^1 Q^T$, where R^1 is easy to compute. The QR factors can provide a more accurate solution of a linear system.

Moreover, if $M \neq n$, the same approach to the rectangular linear system provides a "solution" x that

- if $m > n$, minimizes the norm of the error $||a * x - b||$,
- if $m < n$, of all the multiple solutions to this system, has the minimum norm $||x||$.

If the columns of A are data vectors, then the columns of Q identify the space spanned by the vectors, with the "most important" vectors first, and the diagonal elements of R indicate the new information provided by each successive data vector

The `scipy()` function `qr()` provides the QR factors:

```
from scipy.linalg import qr
Q, R = qr ( A )
```

14 np.linalg.eig(): Eigenvalues

The `numpy.linalg()` function `eig()` returns the eigenvalues `L` and eigenvectors `V` of an $n \times n$ matrix `A`. A matrix :

```
L, V = np.linalg.eig ( A )
AV = np.matmul ( A, V )
VL = np.matmul ( V, np.diag ( L ) )  # Expect AV = VL
```

Writing v_j for column j of `V`, and λ_j for `L[j]`, the j-th eigenpair satisfies

$$A * v_j = \lambda_j * v_j$$

so v_j is a direction in which the influence of A is a simple linear stretching. Except for special cases like symmetric positive definite matrices, there may not be a complete set of eigenpairs; moreover, even if A is real, the eigenpairs may be complex. This means that the eigenvalue information is sometimes awkward to use for analysis. For this reason, the singular value decomposition is often a more useful way to look at what a matrix is doing.

15 `np.linalg.svd()`: The SVD factorization

Let A be an $m \times n$ matrix, square or rectangular, real or complex. There always exists a singular value decomposition of the form

$$A = U \cdot S \cdot V$$

where

- `U` is an $m \times m$ orthogonal or Hermitian matrix of *left singular vectors*;
- `S` is a real, positive $m \times n$ diagonal matrix of *singular values*;
- `V` is an $n \times n$ orthogonal or Hermitian matrix of *right singular vectors*;

For convenience, **`numpy()`** function **`svd`** returns `S` as a vector, which can be restored to matrix form by the **`np.diag()`** function.

```
U, S, V = np.linalg.svd ( A )
USV = np.matmul ( U, np.matmujl ( np.diag ( S ), V )
```

Particular features of the SVD include

- Unlike the eigenvalue decomposition, the SVD always exists as a full factorization of A;
- The factors explain the behavior of A as a linear mapping;
- In particular, the singular values suggest whether some components of A are much more important than others.
- The SVD can be used to solve linear systems;
- The SVD can be used to do least squares solutions of linear systems;
- The SVD can be used to find lower dimensional approximations to A;
- The SVD can be used to find lower dimensional approximations to images, or sets of data.

We will spend a later lesson on the applications of the SVD.