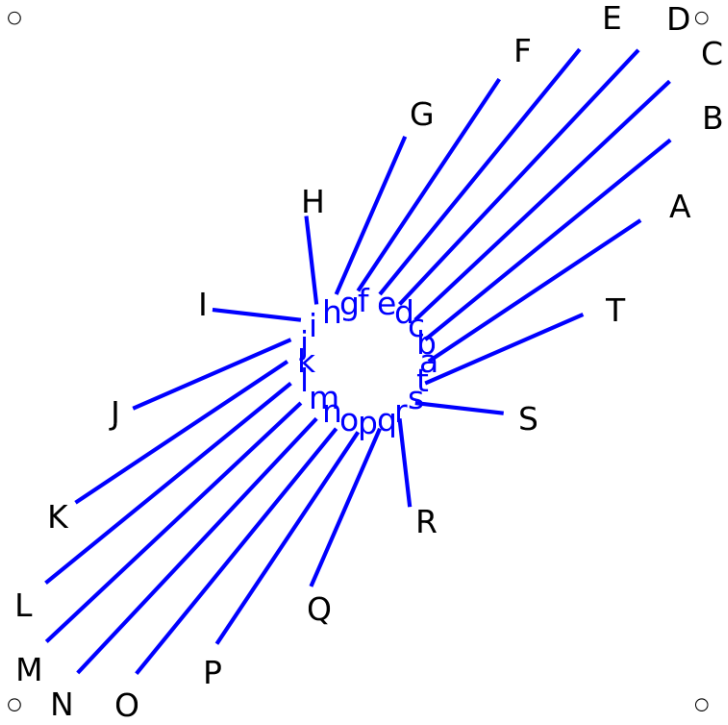
Linear Algebra
Mathematical Programming with Python

MATH 2604: Advanced Scientific Computing 4
Spring 2025
Monday/Wednesday/Friday, 1:00-1:50pm

https://people.sc.fsu.edu/~jburkardt/classes/python_2025/linear_algebra/linear_algebra.pdf

---



How the matrix [4,2;2,4] maps points on the unit circle.

---

1 Linear Algebra

Linear algebra studies the properties of vectors and matrices. It especially considers how a given matrix represents a linear transformation that can be applied to vectors by multiplication, so that $y = A * x$.

Multiplication by A can lengthen or shorten a vector, and change its direction. We are interested in the range of length changes we will see. We may have A and y, and see to solve the corresponding linear system to recover the original vector x.

We may be given many examples of starting vectors x and result vectors y, and seek a matrix A such that the formula $y = A * x$ is approximately true for our data.

2 Matrix properties

Let's consider an example matrix A. For convenience today, we will make sure the matrix is square.

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

There are several matrix properties built into the `linalg()` sublibrary, including

```
np.linalg.det ( A )      # the determinant (0 is singular)
np.linalg.cond ( A )     # the condition (close to 1 is good)
np.linalg.trace ( A )    # sum of diagonal elements
np.diag ( A )            # a vector containing the diagonal of A
```

In our previous discussion of matrices, we also considered commands to compute the sum, minimum, maximum of the whole matrix, the rows, or the columns.

The transpose of a matrix is available:

```
Atran1 = np.transpose ( A )   # call a function
Atran2 = A.T                  # Use the .T operator
```

It is also possible to request the inverse of a matrix, although this is usually a more expensive and inaccurate method to solve systems of equations:

```
Ainv = np.linalg.inv ( A )
```

3 np.diag(), np.tril(), np.triu() to get pieces

If we give the `diag()` command a vector, rather than a matrix, we will compute a new matrix with the given diagonal:

```
d = np.array ( [ 1, 2, 3 ] )
B = np.diag ( d )
```

Sometimes we want to extract the diagonal of a matrix, and make a new diagonal matrix from it.

```
d = np.diag ( A )
C = np.diag ( d )
D = np.diag ( np.diag ( A ) )   # This one line same as previous two
```

We often want to extract the lower or upper triangular part of a matrix. Let the main diagonal have index 0, with the lower and upper diagonals being labeled with negative and positive indices respectively. Then

```
L = np.tril ( A, -1 )
U0 = np.triu ( A, 0 )  #  upper triangle, including diagonal
U1 = np.triu ( A, 1 )  # strict upper triangle
```

4 plt.spy() to display nonzero entries

A matrix in which most entries are nonzero is called "dense". In numerical computations, however, it is common that most matrix entries are actually zero, reflecting the fact that in physics, only the nearby neighbors of a point have a strong influence. A matrix with mostly zero entries is called "sparse". The `spy()` function allows you to see the nonzero pattern of your matrix.

As an example, we can refer to the second difference matrix for a 1-dimensional problem. The entries of this matrix are

$$D2_{i,j} = \begin{cases} -1 & \text{if } j = i-1 \\ +2 & \text{if } j = i \\ -1 & \text{if } j = i+1 \\ 0 & \text{otherwise} \end{cases}$$

To see the result, try the following:

```
from dif2 import dif2
import matplotlib.pyplot as plt
D2 = dif2 ( 10 )
plt.spy ( D2, marker = 'o' )
plt.show ( )
```

5 np.dot() or np.matmul() for multiplication

We have already used the `dot()` function to take the dot product of two vectors.

```
  v1 = np.array ( [ 1, 2, 3 ] )
  v2 = np.array ( [ 3, 4, 5 ] )
  product = np.dot ( v1, v2 )    # returns a number
```

It can also be used for matrix-vector multiplication as in

```
  x = np.array ( [ 1, 2, 3 ] )
  b = np.dot ( A, x )             # returns a vector
  [ 14, 32, 23 ]
```

There is a separate `matmul()` function that can also be used for matrix-vector or matrix-matrix multiplication. If `A` is an $m \times n$ matrix and `B` is an $n \times k$ matrix, we can compute the $m \times k$ product using `matmul()`:

```
  At = A.T                      # At is now an nxm matrix
  AtA = np.matmul ( At, A )     # AtA is an nxn matrix
```

6 `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.

7 np.linalg.norm() for matrix norms

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.

8 Exercise

Suppose that the matrix F and vector y are defined as:

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

Write a Python program in which you:

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

Be sure to use the same type of norm in all your calculations.

9 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`:

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

1. Use this technique to estimate the norm of `G` under each of the l_1, l_2, l_∞ 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.

10 `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 )
```

11 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 )
```

12 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.

13 `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.