NSC1002 Mathematics and Computing: Integrative Tools for Natural Sciences

Professor Beth Wingate

University of Exeter

By the end of this lecture you will…

- Recall that we learned how to use vectors and matrices in our last workshop.
- Learn how to solve systems of algebraic equations using Python.
- Learn how to use numpy linear algebra functions.

Solving systems of linear equations

- You know what systems of linear equations are. An example:

$$\begin{aligned} 4y + 10z &= 20 \\ 2y + \ \ 4z &= \ \ 4 \end{aligned}$$

- You have also learned how to solve them by hand:

$$\underbrace{\begin{pmatrix} 4 & 10 \\ 2 & 4 \end{pmatrix}}_{\mathbf{A}} \underbrace{\begin{pmatrix} y \\ z \end{pmatrix}}_{\mathbf{x}} = \underbrace{\begin{pmatrix} 20 \\ 4 \end{pmatrix}}_{\mathbf{b}} \longrightarrow \begin{aligned} \mathbf{A}^{-1}\mathbf{A}\mathbf{x} &= \mathbf{A}^{-1}\mathbf{b} \\ \mathbf{I}\mathbf{x} &= \mathbf{A}^{-1}\mathbf{b} \\ \mathbf{x} &= \mathbf{A}^{-1}\mathbf{b} \end{aligned}$$

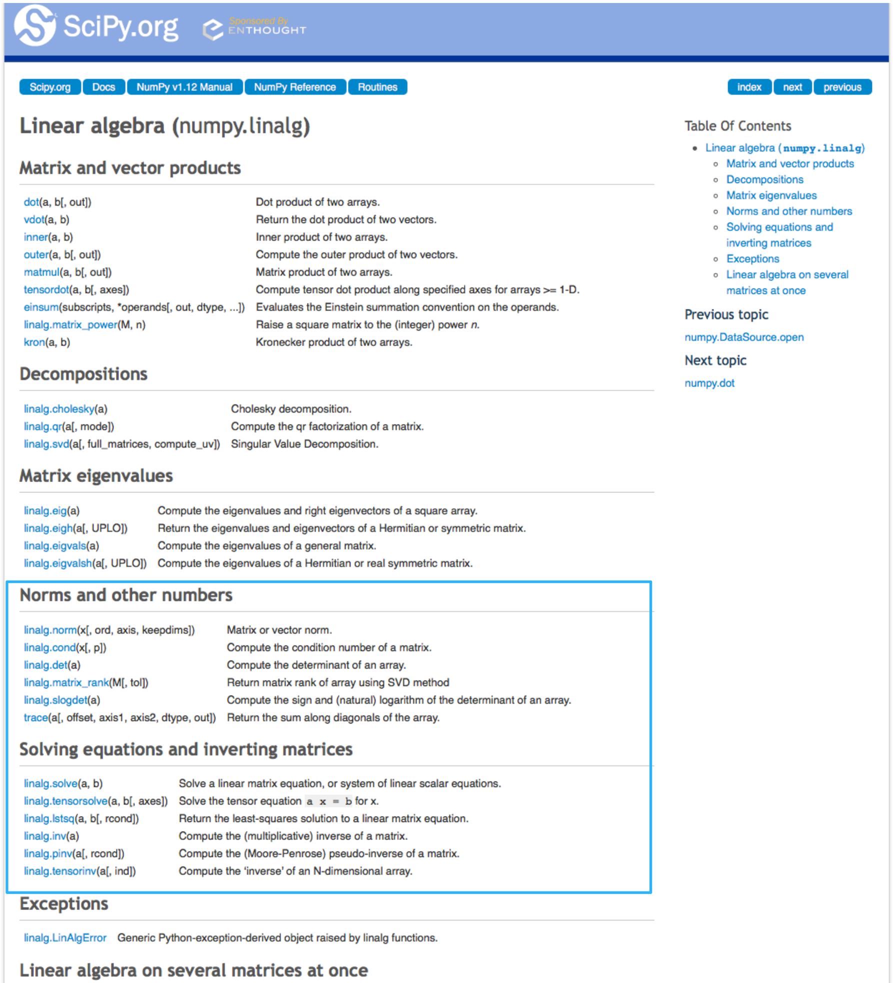
But computing A^{-1} and then, the product $A^{-1}b$ by hand is tough particularly, for large systems.

Python is there to help you!!!!

Solving systems of linear equations with Python

- We need to become acquainted with linalg package of NumPy

https://docs.scipy.org/doc/numpy/reference/routines.linalg.html



SciPy.org Sponsored By ENTHOUGHT

Scipy.org | Docs | NumPy v1.12 Manual | NumPy Reference | Routines | index | next | previous

Linear algebra (numpy.linalg)

Matrix and vector products

dot(a, b[, out])	Dot product of two arrays.
vdot(a, b)	Return the dot product of two vectors.
inner(a, b)	Inner product of two arrays.
outer(a, b[, out])	Compute the outer product of two vectors.
matmul(a, b[, out])	Matrix product of two arrays.
tensordot(a, b[, axes])	Compute tensor dot product along specified axes for arrays >= 1-D.
einsum(subscripts, *operands[, out, dtype, ...])	Evaluates the Einstein summation convention on the operands.
linalg.matrix_power(M, n)	Raise a square matrix to the (integer) power *n*.
kron(a, b)	Kronecker product of two arrays.

Decompositions

linalg.cholesky(a)	Cholesky decomposition.
linalg.qr(a[, mode])	Compute the qr factorization of a matrix.
linalg.svd(a[, full_matrices, compute_uv])	Singular Value Decomposition.

Matrix eigenvalues

linalg.eig(a)	Compute the eigenvalues and right eigenvectors of a square array.
linalg.eigh(a[, UPLO])	Return the eigenvalues and eigenvectors of a Hermitian or symmetric matrix.
linalg.eigvals(a)	Compute the eigenvalues of a general matrix.
linalg.eigvalsh(a[, UPLO])	Compute the eigenvalues of a Hermitian or real symmetric matrix.

Norms and other numbers

linalg.norm(x[, ord, axis, keepdims])	Matrix or vector norm.
linalg.cond(x[, p])	Compute the condition number of a matrix.
linalg.det(a)	Compute the determinant of an array.
linalg.matrix_rank(M[, tol])	Return matrix rank of array using SVD method
linalg.slogdet(a)	Compute the sign and (natural) logarithm of the determinant of an array.
trace(a[, offset, axis1, axis2, dtype, out])	Return the sum along diagonals of the array.

Solving equations and inverting matrices

linalg.solve(a, b)	Solve a linear matrix equation, or system of linear scalar equations.
linalg.tensorsolve(a, b[, axes])	Solve the tensor equation `a x = b` for x.
linalg.lstsq(a, b[, rcond])	Return the least-squares solution to a linear matrix equation.
linalg.inv(a)	Compute the (multiplicative) inverse of a matrix.
linalg.pinv(a[, rcond])	Compute the (Moore-Penrose) pseudo-inverse of a matrix.
linalg.tensorinv(a[, ind])	Compute the 'inverse' of an N-dimensional array.

Exceptions

linalg.LinAlgError	Generic Python-exception-derived object raised by linalg functions.

Linear algebra on several matrices at once

Table Of Contents

Previous topic

numpy.DataSource.open

Next topic

numpy.dot

Solving systems of linear equations with Python

https://docs.scipy.org/doc/numpy/reference/routines.linalg.html

Norms and other numbers

linalg.norm(x[, ord, axis, keepdims])	Matrix or vector norm.
linalg.cond(x[, p])	Compute the condition number of a matrix.
linalg.det(a)	Compute the determinant of an array.
linalg.matrix_rank(M[, tol])	Return matrix rank of array using SVD method
linalg.slogdet(a)	Compute the sign and (natural) logarithm of the determinant of an array.
trace(a[, offset, axis1, axis2, dtype, out])	Return the sum along diagonals of the array.

Solving equations and inverting matrices

linalg.solve(a, b)	Solve a linear matrix equation, or system of linear scalar equations.
linalg.tensorsolve(a, b[, axes])	Solve the tensor equation `a x = b` for x.
linalg.lstsq(a, b[, rcond])	Return the least-squares solution to a linear matrix equation.
linalg.inv(a)	Compute the (multiplicative) inverse of a matrix.
linalg.pinv(a[, rcond])	Compute the (Moore-Penrose) pseudo-inverse of a matrix.
linalg.tensorinv(a[, ind])	Compute the 'inverse' of an N-dimensional array.

Solving systems of linear equations with Python

```
import numpy as np
A = np.array([[4, 10], [2, 4]])
b = np.array([20, 4])
Ainv = np.linalg.inv(A) # Compute the inverse of A
x = np.dot(Ainv, b)      # Compute the unknowns.
print(x)
```

```
[ -10.   6.]
```

$$\mathbf{A}^{-1}\mathbf{A}\mathbf{x} = \mathbf{A}^{-1}\,\mathbf{b}$$

$$\mathbf{I}\mathbf{x} = \mathbf{A}^{-1}\,\mathbf{b}$$

$$\mathbf{x} = \mathbf{A}^{-1}\,\mathbf{b}$$

Solving systems of linear equations with Python

```
import numpy as np
A = np.array([[4, 10], [2, 4]])
b = np.array([20, 4])
Ainv = np.linalg.inv(A) # Compute the inverse of A
x = np.dot(Ainv, b) # Compute the unknowns.
print(x)
```

$$\mathbf{A}^{-1}\mathbf{A}\mathbf{x} = \mathbf{A}^{-1}\,\mathbf{b}$$

$$\mathbf{I}\mathbf{x} = \mathbf{A}^{-1}\,\mathbf{b}$$

$$\mathbf{x} = \mathbf{A}^{-1}\,\mathbf{b}$$

```
[ -10.   6.]
```

But there is a faster way of doing it (using numpy.linalg.solve(a,b))

```
import numpy as np
A = np.array([[4, 10], [2, 4]])
b = np.array([20, 4])
x = np.linalg.solve(A, b)
print(x)
```

```
[ -10.   6.]
```

We get the same answer!

Poorly conditioned matrices: be careful!

- Not all square matrices can be inverted. Matrices with determinant **equal to 0** cannot be inverted (*singular matrices*).

- There are some matrices with a determinant different from 0 but **close to that value** (*poorly-conditioned matrices*). Solving equations whose **A** is poorly-conditioned is problematic (changing **b** only slightly can give us very different results).

- So, in your code, before starting solving a system of linear equations, always check the value of the determinant of **A!!!!!**

Poorly conditioned matrices: be careful!

```
import numpy as np
A = np.array([[4, 10], [2, 4]])
det_a = np.linalg.det(A)
if np.abs(det_a) < .01:
    print("det(A) is very small, solutions may be unreliable!", det_a)
else:
    print("The determinant of matrix A is not close to zero", det_a)
```

```
The determinant of matrix A is not close to zero -4.0
```

Now…

Let’s practice what we have learnt today with the worksheet exercises.