Using Numpy, Linear algebra functionality

Chris Cornwell

Jan 17, 2025

Outline

Outline

- Intro to NumPy
- NumPy arrays
- Linear algebra
- Broadcasting and efficient operations

Getting started with NumPy

- One of the main packages for scientific computing, a must for machine learning and data science.
- Not built-in – must import NumPy into Python session.

Getting started with NumPy

- One of the main packages for scientific computing, a must for machine learning and data science.
- Not built-in – must import NumPy into Python session.

We will also want to track runtime, so we import the package `time`.

Getting started with NumPy

- One of the main packages for scientific computing, a must for machine learning and data science.
- Not built-in – must import NumPy into Python session.

We will also want to track runtime, so we import the package `time`.

```python
import numpy as np
import time
```

Create a shortcut, `np`, for NumPy. This is a common convention.

Getting started with NumPy

- One of the main packages for scientific computing, a must for machine learning and data science.
- Not built-in – must import NumPy into Python session.

We will also want to track runtime, so we import the package `time`.

```
import numpy as np
import time
```

Create a shortcut, `np`, for NumPy. This is a common convention.

- Depending on how you are interacting with Python, may have to *install* the `numpy` package before the first use. Open a command terminal (`Ctrl+``, in VSCode on Windows) and type the appropriate command below.

  `py -m pip install numpy` (Windows)

  `python3 -m pip install numpy` (macOS)

  `sudo pip install numpy` (Linux based)

When installing other packages, replace `numpy` with the package name. After install, the import commands above should run without error.

Outline

Basic NumPy arrays

The main type of object in NumPy is the `ndarray` (n-dimensional array), which is constructed from a list using the command

`np.array(the_list)`.

Basic NumPy arrays

The main type of object in NumPy is the `ndarray` (n-dimensional array), which is constructed from a list using the command

```
np.array(the_list).
```

If items in `the_list` are of numeric type, then think of the resulting `ndarray` as like a vector. Operations on NumPy arrays work like vectors in linear algebra.

Basic NumPy arrays

The main type of object in NumPy is the `ndarray` (n-dimensional array), which is constructed from a list using the command

`np.array(the_list).`

If items in `the_list` are of numeric type, then think of the resulting `ndarray` as like a vector. Operations on NumPy arrays work like vectors in linear algebra.

Example:

```
v = np.array([-1, 1, 1])
w = np.array([0.5, 0, 1.1])
# print the (vector) sum: [-0.5  1.   2.1]
print(v + w)
# prints [1.0, 0.0, 2.2]
print(2*w)
```

More than 1d

A NumPy array from a list containing numeric types makes a vector – also known as a 1-*dimensional array* (Python language), or a *tensor of order* 1 (mathematics).

More than `1d`

A NumPy array from a list containing numeric types makes a vector – also known as a 1-*dimensional array* (Python language), or a *tensor of order* 1 (mathematics).

A 2-dimensional array, or tensor of order 2, is like a matrix. You construct it with `np.array()` from a list of lists – each of the same length.

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

Each “inside list” is a row. The array `A` is a 2×3 matrix.

More than 1d

A NumPy array from a list containing numeric types makes a vector – also known as a 1-*dimensional array* (Python language), or a *tensor of order* 1 (mathematics).

A 2-dimensional array, or tensor of order 2, is like a matrix. You construct it with `np.array()` from a list of lists – each of the same length.

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

Each "inside list" is a row. The array `A` is a 2×3 matrix.

Every array in NumPy has an attribute `shape`.

- Previous slide: `v = np.array([-1,1,1])` has `v.shape = (3,)`.
- The matrix `A`: `A.shape` is equal to `(2, 3)`.

Operations on arrays

Multiplying two arrays: most recent version of Python uses the `@` symbol.[1] When the arrays are both matrices, it computes their matrix product; when one is a vector, it computes the matrix-vector product; when both are vectors, it computes the dot product.

[1] In older versions, matrix multiplication is `np.matmul()` and dot product is `np.dot()`.

Operations on arrays

Multiplying two arrays: most recent version of Python uses the `@` symbol.[1] When the arrays are both matrices, it computes their matrix product; when one is a vector, it computes the matrix-vector product; when both are vectors, it computes the dot product.

For example, say that `A` is the matrix $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ from before, `v` is the vector $(-1, 1, 1)$, and let `B` and `u` be the matrix and vector defined in the code below.

```
B = np.array([[1, 0], [1, -1], [1, 1]])
u = np.array([1, 1, 0])

(A @ B, A @ v, v @ u)
```

[1] In older versions, matrix multiplication is `np.matmul()` and dot product is `np.dot()`.

Operations on arrays

Multiplying two arrays: most recent version of Python uses the `@` symbol.[1] When the arrays are both matrices, it computes their matrix product; when one is a vector, it computes the matrix-vector product; when both are vectors, it computes the dot product.

For example, say that `A` is the matrix $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ from before, `v` is the vector $(-1, 1, 1)$, and let `B` and `u` be the matrix and vector defined in the code below.

```
B = np.array([[1, 0], [1, -1], [1, 1]])
u = np.array([1, 1, 0])

(A @ B, A @ v, v @ u)
```

- Output is the ordered triple

  (array([[6, 1], [15, 1]]), array([4, 7]), 0).

[1]In older versions, matrix multiplication is `np.matmul()` and dot product is `np.dot()`.

Indexing and slicing arrays

Items in 1d array are accessed the same way as in a list

e.g., `v[0]` is the first item, at index 0.

For a 2d array, say the matrix `A`, we can access the item in the row `i` and column `j` by `A[i, j]`.

[2] Recall the use of the colon from before. It functions the same way here.

Indexing and slicing arrays

Items in 1d array are accessed the same way as in a list

e.g., `v[0]` is the first item, at index 0.

For a 2d array, say the matrix `A`, we can access the item in the row `i` and column `j` by `A[i, j]`.

Like lists, can also use slicing with arrays. Examples:

1. To get first column of the matrix `A`, write `A[:, 0]`.[2]

[2] Recall the use of the colon from before. It functions the same way here.

Indexing and slicing arrays

Items in 1d array are accessed the same way as in a list

e.g., `v[0]` is the first item, at index 0.

For a 2d array, say the matrix `A`, we can access the item in the row `i` and column `j` by `A[i, j]`.

Like lists, can also use slicing with arrays. Examples:

1. To get first column of the matrix `A`, write `A[: , 0]`.[2]
2. To get top-right 2×2 submatrix of a matrix `M`, then use `M[:2, -2:]`.

[2] Recall the use of the colon from before. It functions the same way here.

Indexing and slicing arrays

Items in 1d array are accessed the same way as in a list

e.g., `v[0]` is the first item, at index 0.

For a 2d array, say the matrix `A`, we can access the item in the row `i` and column `j` by `A[i, j]`.

Like lists, can also use slicing with arrays. Examples:

1. To get first column of the matrix `A`, write `A[:, 0]`.[2]
2. To get top-right 2×2 submatrix of a matrix `M`, then use `M[:2, -2:]`.

With arrays, can even get non-consecutive indices. For example, `A[:,[0,2]]` gives two columns that are not adjacent.

[2] Recall the use of the colon from before. It functions the same way here.

Indexing and slicing arrays

Items in 1d array are accessed the same way as in a list

e.g., `v[0]` is the first item, at index 0.

For a 2d array, say the matrix `A`, we can access the item in the row `i` and column `j` by `A[i, j]`.

Like lists, can also use slicing with arrays. Examples:

1. To get first column of the matrix `A`, write `A[: , 0]`.[2]
2. To get top-right 2×2 submatrix of a matrix `M`, then use `M[:2, -2:]`.

With arrays, can even get non-consecutive indices. For example, `A[: ,[0,2]]` gives two columns that are not adjacent.
If `A` is a 2d array, its transpose is `A.T` (providing yet another alternative for accessing a column).

[2] Recall the use of the colon from before. It functions the same way here.

Outline

Constructing special matrices

Some types of matrices are used a lot; would be cumbersome to always write the row lists ourselves (e.g., in a 100×100 matrix).

> **Zero matrix**: The command `np.zeros((m, n))` constructs an $m \times n$ matrix with all entries equal to zero.

Constructing special matrices

Some types of matrices are used a lot; would be cumbersome to always write the row lists ourselves (e.g., in a 100×100 matrix).

Zero matrix: The command `np.zeros((m, n))` constructs an $m \times n$ matrix with all entries equal to zero.

Diagonal matrix: If `d` is a 1d array of length n, the command `np.diag(d)` constructs an $n \times n$ diagonal matrix which has `d` as its diagonal entries.

Constructing special matrices

Some types of matrices are used a lot; would be cumbersome to always write the row lists ourselves (e.g., in a 100×100 matrix).

Zero matrix: The command `np.zeros((m, n))` constructs an $m \times n$ matrix with all entries equal to zero.

Diagonal matrix: If `d` is a 1d array of length n, the command `np.diag(d)` constructs an $n \times n$ diagonal matrix which has `d` as its diagonal entries.

Identity matrix: The command `np.identity(n)` (also, `np.eye(n)`) constructs the $n \times n$ identity matrix.

Extracting part of matrix: May want to get part of a matrix. To get a submatrix from consecutive rows and columns, use slicing.

Constructing special matrices

Some types of matrices are used a lot; would be cumbersome to always write the row lists ourselves (e.g., in a 100×100 matrix).

Zero matrix: The command `np.zeros((m, n))` constructs an $m \times n$ matrix with all entries equal to zero.

Diagonal matrix: If `d` is a 1d array of length n, the command `np.diag(d)` constructs an $n \times n$ diagonal matrix which has `d` as its diagonal entries.

Identity matrix: The command `np.identity(n)` (also, `np.eye(n)`) constructs the $n \times n$ identity matrix.

Extracting part of matrix: May want to get part of a matrix. To get a submatrix from consecutive rows and columns, use slicing. Also, here are functions that return part of the matrix (other entries being set to 0).

```
# return lower triangular part (at or below the diagonal)
np.tril(A)
# return upper triangular part (at or above the diagonal)
np.triu(A)
# return the diagonal of A
np.diag(A)
```

Using NumPy for linear algebra

In addition to the product operations on arrays, NumPy has a library (`linalg`) with many functions for linear algebra.

Using NumPy for linear algebra

In addition to the product operations on arrays, NumPy has a library (`linalg`) with many functions for linear algebra.
Examples:

1. If M is a square matrix, can compute $\det(M)$ with the command `np.linalg.det(M)`.

Using NumPy for linear algebra

In addition to the product operations on arrays, NumPy has a library (`linalg`) with many functions for linear algebra.
Examples:

1. If M is a square matrix, can compute $\det(M)$ with the command `np.linalg.det(M)`.
2. When M is invertible, can compute M^{-1} with the command `np.linalg.inv(M)`.

Using NumPy for linear algebra

In addition to the product operations on arrays, NumPy has a library (`linalg`) with many functions for linear algebra.
Examples:

1. If M is a square matrix, can compute $\det(M)$ with the command `np.linalg.det(M)`.
2. When M is invertible, can compute M^{-1} with the command `np.linalg.inv(M)`.
3. If M is a square matrix, can compute eigenvalues and eigenvectors with `np.linalg.eig(M)`.

Using NumPy for linear algebra

In addition to the product operations on arrays, NumPy has a library (`linalg`) with many functions for linear algebra.
Examples:

1. If M is a square matrix, can compute $\det(M)$ with the command `np.linalg.det(M)`.
2. When M is invertible, can compute M^{-1} with the command `np.linalg.inv(M)`.
3. If M is a square matrix, can compute eigenvalues and eigenvectors with `np.linalg.eig(M)`.

There are many other linear algebra functions (see the docs here). Some are only implemented for square matrices (and perhaps only invertible ones), even though it would make sense to have them work more generally – for example, `np.solve(A, b)` only solves the system $A\mathbf{x} = b$ if `A` is a square invertible matrix.

Solving a linear system & Errors

To solve $A\mathbf{x} = \mathbf{b}$, with a square invertible matrix `A` and vector `b` of the right size, you can use `np.linalg.solve(A, b)`.

[3]While writing this slide, Github Copilot suggested I write that it would be a `ValueError` from *mismatched dimensions*: rows of `A` being size `(3,)` and the vector `b` being size `(2,)`.

Solving a linear system & Errors

To solve $A\mathbf{x} = \mathbf{b}$, with a square invertible matrix `A` and vector `b` of the right size, you can use `np.linalg.solve(A, b)`.
What happens when `A` is not square? Execute the following code in Python.

```
A = np.array([[1, 2, 3], [1, 4, -1]])
b = np.array([1, -5])
# system has solution x = [0, -1, 1]
# but next line raises an error
x = np.linalg.solve(A, b)
```

[3]While writing this slide, Github Copilot suggested I write that it would be a `ValueError` from *mismatched dimensions*: rows of `A` being size `(3,)` and the vector `b` being size `(2,)`.

Solving a linear system & Errors

To solve $A\mathbf{x} = \mathbf{b}$, with a square invertible matrix `A` and vector `b` of the right size, you can use `np.linalg.solve(A, b)`.
What happens when `A` is not square? Execute the following code in Python.

```
A = np.array([[1, 2, 3], [1, 4, -1]])
b = np.array([1, -5])
# system has solution x = [0, -1, 1]
# but next line raises an error
x = np.linalg.solve(A, b)
```

A message is generated about the error. It gives you helpful information, if it can. In this case, it is a `LinAlgError` with the message `Last 2 dimensions of the array must be square.`

[3]While writing this slide, Github Copilot suggested I write that it would be a `ValueError` from *mismatched dimensions*: rows of `A` being size `(3,)` and the vector `b` being size `(2,)`.

Solving a linear system & Errors

To solve $A\mathbf{x} = \mathbf{b}$, with a square invertible matrix `A` and vector `b` of the right size, you can use `np.linalg.solve(A, b)`.
What happens when `A` is not square? Execute the following code in Python.

```
A = np.array([[1, 2, 3], [1, 4, -1]])
b = np.array([1, -5])
# system has solution x = [0, -1, 1]
# but next line raises an error
x = np.linalg.solve(A, b)
```

A message is generated about the error. It gives you helpful information, if it can. In this case, it is a `LinAlgError` with the message `Last 2 dimensions of the array must be square.`
Spend time trying to use error messages to understand issues in your code. Also, have healthy skepticism about AI assistants. They hallucinate; error messages don't.[3]

[3]While writing this slide, Github Copilot suggested I write that it would be a `ValueError` from *mismatched dimensions*: rows of `A` being size `(3,)` and the vector `b` being size `(2,)`.

Outline

Broadcasting, universal functions

Say that you have a 1d array and you want to make array with square root the entries.

> First thought: use a loop, taking square root (and assigning) as you go through items in the array.

NumPy has an efficient way to handle it, called *broadcasting*. If `v` is your array, then you can simply type

```
sqrt_v = np.sqrt(v)
```

> The function `np.sqrt()` takes the square root of each entry in `v`; you don't need to write the for loop.[4]

Functions that work on arrays this way are quite common in NumPy. They are called **ufuncs** (universal functions).

[4]Technically, there's a for loop in the background, but it happens in C and works much faster.

Broadcasting, universal functions

Say that you have a 1d array and you want to make array with square root the entries.

First thought: use a loop, taking square root (and assigning) as you go through items in the array.

NumPy has an efficient way to handle it, called *broadcasting*. If `v` is your array, then you can simply type

```
sqrt_v = np.sqrt(v)
```

The function `np.sqrt()` takes the square root of each entry in `v`; you don't need to write the for loop.[4]

Functions that work on arrays this way are quite common in NumPy. They are called **ufuncs** (universal functions).
Other examples of ufuncs in NumPy:

`np.abs()`, `np.sum()`, `np.maximum()`, `np.minimum()`, `np.exp()`, `np.log()`.

[4]Technically, there's a for loop in the background, but it happens in C and works much faster.

More on broadcasting

Many basic operations with NumPy arrays use broadcasting. Here are a few examples with an array `v`.

1. To add the same scalar, say 3, to every array entry: type `v+3`.
2. To multiply every entry by 3: type `3*v`.

[5] In mathematics, this product on vectors is called the Hadamard product.

More on broadcasting

Many basic operations with NumPy arrays use broadcasting. Here are a few examples with an array `v`.

1. To add the same scalar, say 3, to every array entry: type `v+3`.
2. To multiply every entry by 3: type `3*v`.
3. To square every entry of an array: type `v**2`.
4. To multiply `v` by another array `w`, entry-wise[5]: type `v*w`.

[5] In mathematics, this product on vectors is called the Hadamard product.

More on broadcasting

Many basic operations with NumPy arrays use broadcasting. Here are a few examples with an array `v`.

1. To add the same scalar, say 3, to every array entry: type `v+3`.
2. To multiply every entry by 3: type `3*v`.
3. To square every entry of an array: type `v**2`.
4. To multiply `v` by another array `w`, entry-wise[5]: type `v*w`.

Everything mentioned here works just as well on matrices (2d arrays), and generally on any `nd` array (higher order tensors).

[5] In mathematics, this product on vectors is called the Hadamard product.

More on broadcasting

Many basic operations with NumPy arrays use broadcasting. Here are a few examples with an array `v`.

1. To add the same scalar, say 3, to every array entry: type `v+3`.
2. To multiply every entry by 3: type `3*v`.
3. To square every entry of an array: type `v**2`.
4. To multiply `v` by another array `w`, entry-wise[5]: type `v*w`.

Everything mentioned here works just as well on matrices (2d arrays), and generally on any `nd` array (higher order tensors).

Exercise.
Write out code that uses broadcasting to create a 100×100 matrix where all non-diagonal entries are -1 and all diagonal entries are 2.

[5] In mathematics, this product on vectors is called the Hadamard product.

Experiment with runtime for universal function

To check the efficiency of broadcasting, use the `time` package. Beforehand, make sure that you imported both `numpy` and `time` (see slide in first section).

Experiment with runtime for universal function

To check the efficiency of broadcasting, use the `time` package. Beforehand, make sure that you imported both `numpy` and `time` (see slide in first section).

Something simple: from a large identity matrix, we will get the exponential of the matrix (apply the function e^x to every entry).

Experiment with runtime for universal function

To check the efficiency of broadcasting, use the `time` package.
Beforehand, make sure that you imported both `numpy` and `time` (see slide in first section).
Something simple: from a large identity matrix, we will get the exponential of the matrix (apply the function e^x to every entry).
First, we use a `for` loop. Run the code below in your Jupyter notebook.

```
id_matrix = np.eye(1000)
exp_matrix = np.zeros((1000, 1000))
start = time.time()
for i in range(1000):
    for j in range(1000):
        exp_matrix[i,j] = np.exp(id_matrix[i,j])
end = time.time()
print(f"Seconds taken: {end-start}.")
```

Experiment with runtime for universal function

To check the efficiency of broadcasting, use the `time` package. Beforehand, make sure that you imported both `numpy` and `time` (see slide in first section).
Something simple: from a large identity matrix, we will get the exponential of the matrix (apply the function e^x to every entry).
First, we use a `for` loop. Run the code below in your Jupyter notebook.

```
id_matrix = np.eye(1000)
exp_matrix = np.zeros((1000, 1000))
start = time.time()
for i in range(1000):
    for j in range(1000):
        exp_matrix[i,j] = np.exp(id_matrix[i,j])
end = time.time()
print(f"Seconds taken: {end-start}.")
```

The output gives the number of seconds to run the computation. The exact time will vary based on your computer. Mine took around 0.55 seconds.

Experiment with runtime for universal function

Now, we will use broadcasting to compute the exponential of the identity matrix.

Experiment with runtime for universal function

Now, we will use broadcasting to compute the exponential of the identity matrix.

Run the following code in your Jupyter notebook.

```
id_matrix = np.eye(1000)
exp_matrix = np.zeros((1000, 1000))
start = time.time()
exp_matrix = np.exp(id_matrix)
end = time.time()
print(f"Seconds taken: {end-start}.")
```

Experiment with runtime for universal function

Now, we will use broadcasting to compute the exponential of the identity matrix.
Run the following code in your Jupyter notebook.

```
id_matrix = np.eye(1000)
exp_matrix = np.zeros((1000, 1000))
start = time.time()
exp_matrix = np.exp(id_matrix)
end = time.time()
print(f"Seconds taken: {end-start}.")
```

Again, the output is the number of seconds of runtime. For this approach with `np.exp()`, my computer took around 0.0045 seconds. That is over 100 times faster than writing the loop!