AA 274
Principles of Robotic Autonomy

Tutorial 3: Scientific Computing in Python

Boris Ivanovic

Stanford University

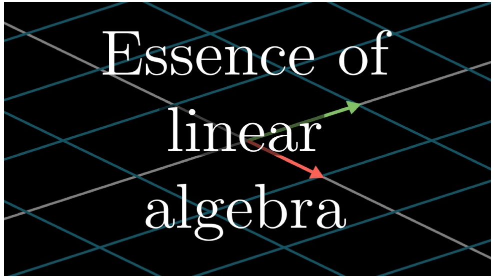
Autonomous Systems Lab

Overview for Today

Linear Algebra Review

\+

Linear Algebra in Python

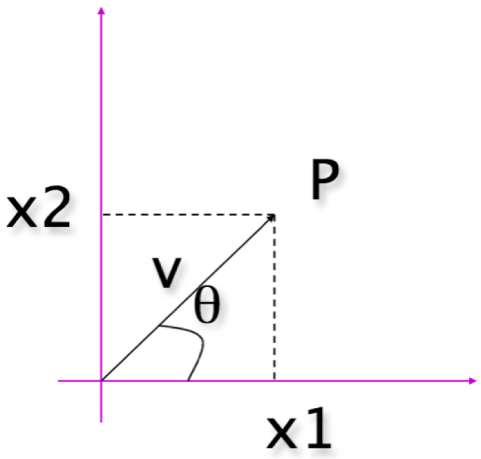


Introduction to Jupyter Notebooks

\+

(Brief) Introduction to TensorFlow

Vector Review



$\boldsymbol{v} = (x_1, x_2)$

Magnitude: $||\boldsymbol{v}|| = \sqrt{x_1^2 + x_2^2}$

If $||\boldsymbol{v}|| = 1$, then $\boldsymbol{v}$ is a **UNIT** vector. E.g. $\frac{\boldsymbol{v}}{||\boldsymbol{v}||} = \left(\frac{x_1}{||\boldsymbol{v}||}, \frac{x_2}{||\boldsymbol{v}||}\right)$ is a unit vector.

Orientation: $\theta = \tan^{-1}\left(\frac{x_1}{x_2}\right)$

Vector Review

$$\boldsymbol{v} + \boldsymbol{w} = (x_1, x_2) + (y_1, y_2) = (x_1 + y_1, x_2 + y_2)$$

$$\boldsymbol{v} - \boldsymbol{w} = (x_1, x_2) - (y_1, y_2) = (x_1 - y_1, x_2 - y_2)$$

$$a\boldsymbol{v} = a(x_1, x_2) = (ax_1, ax_2)$$

Matrix Review

$$A_{n\times m} = \begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{bmatrix}$$

Sum: $C_{n\times m} = A_{n\times m} + B_{n\times m} \Rightarrow c_{ij} = a_{ij} + b_{ij}$

A and B must have the same dimensions!

Example: $\begin{bmatrix} 2 & 5 \\ 3 & 1 \end{bmatrix} + \begin{bmatrix} 6 & 2 \\ 1 & 5 \end{bmatrix} = \begin{bmatrix} 8 & 7 \\ 4 & 6 \end{bmatrix}$

Matrices and Vectors (In Python)

```
import numpy as np
```

A supremely-optimized, well-maintained scientific computing package for Python.

As time goes on, you'll learn to appreciate NumPy more and more.

Years later, I'm **still** learning new things about it!

Matrices and Vectors (In Python)

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \quad v = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

```
import numpy as np

M = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])

v = np.array([[1],
              [2],
              [3]])
```

***Never** use `np.matrix`, stay consistent and only use `np.array`.

Matrices and Vectors (in Python, cont'd)

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \quad v = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

```
print M.shape
>>> (3, 3)

print v.shape
>>> (3, 1)

v_single_dim = np.array([1, 2, 3])
print v_single_dim.shape
>>> (3,)
```

Matrices and Vectors (in Python, cont'd)

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \quad v = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

```
print v + v
>>> [[2]
     [4]
     [6]]

print 3*v
>>> [[3]
     [6]
     [9]]
```

Other Ways to Create Matrices and Vectors

NumPy provides many convenience functions for creating matrices/vectors.

```
a = np.zeros((2,2))   # Create an array of all zeros
print a               # Prints "[[ 0.  0.]
                      #          [ 0.  0.]]"

b = np.ones((1,2))    # Create an array of all ones
print b               # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7) # Create a constant array
print c                # Prints "[[ 7.  7.]
                       #          [ 7.  7.]]"

d = np.eye(2)         # Create a 2x2 identity matrix
print d               # Prints "[[ 1.  0.]
                      #          [ 0.  1.]]"

e = np.random.random((2,2)) # Create an array filled with random values
print e                     # Might print "[[ 0.91940167  0.08143941]
                            #               [ 0.68744134  0.87236687]]"
```

Other Ways to Create Matrices and Vectors (cont'd)

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \quad v = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

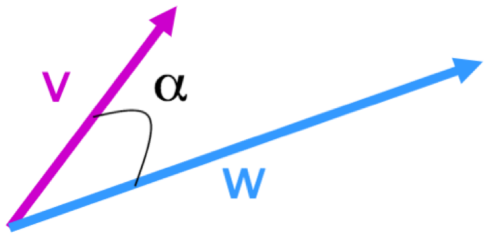
```
v1 = np.array([1, 2, 3])
v2 = np.array([4, 5, 6])
v3 = np.array([7, 8, 9])
M = np.vstack([v1, v2, v3])
print M
>>> [[1 2 3]
     [4 5 6]
     [7 8 9]]

# There is also a way to do this horizontally =>
hstack
```

Matrix Indexing

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \quad v = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

```
print M
>>> [[1 2 3]
     [4 5 6]
     [7 8 9]]

print M[:2, 1:3]
>>> [[2 3]
     [5 6]]
```

Dot Product



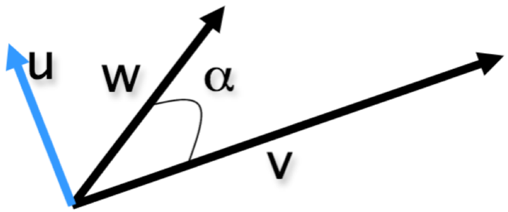
$$v \cdot w = (x_1, x_2) \cdot (y_1, y_2) = x_1 y_1 + x_2 y_2$$

The inner product is a **SCALAR!**

$$v \cdot w = (x_1, x_2) \cdot (y_1, y_2) = ||v|| \cdot ||w|| \cos(\alpha)$$

If $v \perp w$, then $v \cdot w = 0$

Cross Product



$u = v \times w$

The cross product is a **VECTOR!**

Magnitude: $||u|| = ||v \times w|| = ||v|| \cdot ||w|| \sin(\alpha)$

Orientation:
$$u \perp v \Rightarrow u \cdot v = (v \times w) \cdot v = 0$$
$$u \perp w \Rightarrow u \cdot w = (v \times w) \cdot w = 0$$

If $v \,//\, w$, then $\mathrm{u} = 0$

Cross Product

$$\boldsymbol{i} = (1,0,0) \qquad ||\boldsymbol{i}|| = 1 \qquad \boldsymbol{i} = \boldsymbol{j}\times\boldsymbol{k}$$

$$\boldsymbol{j} = (0,1,0) \qquad ||\boldsymbol{j}|| = 1 \qquad \boldsymbol{j} = \boldsymbol{k}\times\boldsymbol{i}$$

$$\boldsymbol{k} = (0,0,1) \qquad ||\boldsymbol{k}|| = 1 \qquad \boldsymbol{k} = \boldsymbol{i}\times\boldsymbol{j}$$

$$\boldsymbol{u} = \boldsymbol{v}\times\boldsymbol{w} = (x_1, x_2, x_3)\times(y_1, y_2, y_3)$$
$$= (x_2y_3 - x_3y_2)\boldsymbol{i} + (x_3y_1 - x_1y_3)\boldsymbol{j} + (x_1y_2 - x_2y_1)\boldsymbol{k}$$

Matrix Multiplication

$$A_{n\times m} = \begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{bmatrix} \quad \boldsymbol{a_i}$$

$$B_{m\times p} = \begin{bmatrix} b_{11} & \cdots & b_{1p} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{mp} \end{bmatrix}$$

$$\boldsymbol{b_j}$$

Product:

$$C_{n\times p} = A_{n\times \boxed{m}} B_{\boxed{m}\times p}$$

$$c_{ij} = \boldsymbol{a_i} \cdot \boldsymbol{b_j} = \sum_{k=1}^{m} a_{ik} b_{kj}$$

A and B must have compatible dimensions!

$$A_{n\times n} B_{n\times n} \neq B_{n\times n} A_{n\times n}$$

Basic Operations – Dot Multiplication

$$M = \begin{bmatrix} 3 & 0 & 2 \\ 2 & 0 & -2 \\ 0 & 1 & 1 \end{bmatrix}, \quad v = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

```
print M.dot(v)
>>> [[ 9]
     [-4]
     [ 5]]

print v.dot(v)
>>> ValueError: shapes (3,1) and (3,1) not
    aligned: 1 (dim 1) != 3 (dim 0)

print v.T.dot(v)
>>> [[14]] # Why these brackets? Because it’s (1,1)-shaped
```

Basic Operations - Cross Multiplication

$$v_1 = \begin{bmatrix} 3 \\ -3 \\ 1 \end{bmatrix}, \quad v_2 = \begin{bmatrix} 4 \\ 9 \\ 2 \end{bmatrix}$$

```
print v1.cross(v2)
>>> Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
    AttributeError: 'numpy.ndarray' object has no attribute
    'cross'

# Yeah... Slightly convoluted because np.cross() assumes
# horizontal vectors.
print np.cross(v1, v2, axisa=0, axisb=0).T
>>> [[-15]
     [ -2]
     [ 39]]
```

Basic Operations - Element-wise Multiplication

$$M = \begin{bmatrix} 3 & 0 & 2 \\ 2 & 0 & -2 \\ 0 & 1 & 1 \end{bmatrix}, \quad v = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

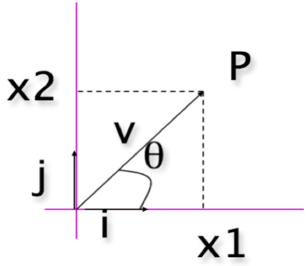
```
print np.multiply(M, v)
>>> [[ 3   0   2]
     [ 4   0  -4]
     [ 0   3   3]]

print np.multiply(v, v)
>>> [[1]
     [4]
     [9]]
```

This works because of something called **broadcasting** where NumPy will implicitly replicate arrays across dimensions that aren't the same.

https://docs.scipy.org/doc/numpy-1.15.0/user/basics.broadcasting.html

Orthonormal Basis

= Orthogonal and Normalized Basis



$$\boldsymbol{i} = (1{,}0) \quad ||i|| = 1$$

$$\boldsymbol{j} = (0{,}1) \quad ||j|| = 1 \qquad (\boldsymbol{i} \cdot \boldsymbol{j} = 0)$$

$$\boldsymbol{v} = (x_1, x_2) \qquad \boldsymbol{v} = x_1\boldsymbol{i} + x_2\boldsymbol{j}$$

$$\boldsymbol{v} \cdot \boldsymbol{i} = (x_1\boldsymbol{i} + x_2\boldsymbol{j}) \cdot \boldsymbol{i} = x_1 1 + x_2 0 = x_1$$

$$\boldsymbol{v} \cdot \boldsymbol{j} = (x_1\boldsymbol{i} + x_2\boldsymbol{j}) \cdot \boldsymbol{j} = x_1 0 + x_2 1 = x_2$$

Transpose

Definition: $\boldsymbol{C}_{m\times n} = \boldsymbol{A}^T_{n\times m}$ $\quad c_{ij} = a_{ji}$

Identities: $(\boldsymbol{A}+\boldsymbol{B})^T = \boldsymbol{A}^T + \boldsymbol{B}^T$ $\quad (\boldsymbol{AB})^T = \boldsymbol{B}^T\boldsymbol{A}^T$

If $\boldsymbol{A} = \boldsymbol{A}^T$, then $\boldsymbol{A}$ is *symmetric*.

Basic Operations – Transpose

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \quad v = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

```
print M.T
>>> [[1 4 7]
     [2 5 8]
     [3 6 9]]

print v.T
>>> [[1 2 3]]

print M.T.shape, v.T.shape
>>> (3, 3) (1, 3)
```

Matrix Determinant

Useful value computed from the elements of a *square* matrix $\boldsymbol{A}$.

$$\det[a_{11}] = a_{11}$$

$$\det\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = a_{11}a_{22} - a_{12}a_{21}$$

$$\det\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32}$$

$$-a_{13}a_{22}a_{31} - a_{23}a_{32}a_{11} - a_{33}a_{12}a_{21}$$

Matrix Inverse

Does not exist for all matrices, necessary (but not sufficient) that the matrix is square

$$\boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{A}^{-1}\boldsymbol{A} = \boldsymbol{I}$$

$$\boldsymbol{A}^{-1} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}^{-1} = \frac{1}{\det(\boldsymbol{A})} \begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix}, \qquad \det(\boldsymbol{A}) \neq 0$$

If $\det(\boldsymbol{A}) = 0$, then $\boldsymbol{A}$ does not have an inverse.

Basic Operations - Determinant and Inverse

$$M = \begin{bmatrix} 3 & 0 & 2 \\ 2 & 0 & -2 \\ 0 & 1 & 1 \end{bmatrix}, \quad v = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

```
print np.linalg.inv(M)
>>> [[ 0.2  0.2  0. ]
     [-0.2  0.3  1. ]
     [ 0.2 -0.3 -0. ]]

Be careful of matrices that are not invertible!

print np.linalg.det(M)
>>> 10.0 # Thankfully ours is.
```

Matrix Eigenvalues and Eigenvectors

An eigenvalue λ and eigenvector $\boldsymbol{u}$ satisfies

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

where $\boldsymbol{A}$ is a square matrix.

=> Multiplying $\boldsymbol{u}$ by $\boldsymbol{A}$ scales $\boldsymbol{u}$ by λ

Matrix Eigenvalues and Eigenvectors

Rearranging the previous equation gives the form

$$\boldsymbol{A}\boldsymbol{u} - \lambda\boldsymbol{u} = (\boldsymbol{A} - \lambda\boldsymbol{I})\boldsymbol{u} = 0$$

which has a solution if and only if $\det(\boldsymbol{A} - \lambda\boldsymbol{I}) = 0$

=> The eigenvalues are roots of this determinant which is polynomial in λ.

=> Substitute the resulting eigenvalues back into $\boldsymbol{A}\boldsymbol{u} = \lambda\boldsymbol{u}$ and solve to obtain the corresponding eigenvector.

Basic Operations - Eigenvalues, Eigenvectors

$$M = \begin{bmatrix} 0 & 1 \\ -2 & -3 \end{bmatrix}$$

```
eigvals, eigvecs = np.linalg.eig(M)
print eigvals
>>> [-1. -2.]

print eigvecs
>>> [[ 0.70710678 -0.4472136 ]
     [-0.70710678  0.89442719]]
```

NOTE: Please read the NumPy docs on this function before using it, lots more information about multiplicity of eigenvalues and etc there:

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

Singular Value Decomposition

Singular values: Non-negative square roots of the eigenvalues of $\boldsymbol{A}^T\boldsymbol{A}$. Denoted as $\sigma_i, i = 1, \ldots, n$

SVD: If $\boldsymbol{A}$ is a real $m \times n$ matrix, then there exist orthogonal matrices $\boldsymbol{U}$ ($\in \mathbb{R}^{m\times m}$) and $\boldsymbol{V}$ ($\in \mathbb{R}^{n\times n}$) such that:

$$\boldsymbol{A} = \boldsymbol{U\Sigma}\mathbf{V}^{-1} \text{ and } \boldsymbol{U}^{-1}\boldsymbol{AV} = \boldsymbol{\Sigma} = \begin{bmatrix} \sigma_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma_n \end{bmatrix}$$

Singular Value Decomposition

Suppose we know the singular values of $\boldsymbol{A}$ and we know r are non-zero

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r \geq \sigma_{r+1} = \cdots = \sigma_p = 0$$

- Rank($\boldsymbol{A}$) = r
- Null($\boldsymbol{A}$) = span$\{\boldsymbol{v_{r+1}}, \dots, \boldsymbol{v_n}\}$
- Range($\boldsymbol{A}$) = span$\{\boldsymbol{u_1}, \dots, \boldsymbol{u_r}\}$

$\left||\boldsymbol{A}|\right|_{F^2} = \sigma_1^2 + \sigma_2^2 + \cdots + \sigma_p^2$ $\qquad$ $\left||\boldsymbol{A}|\right|_2 = \sigma_1$

Numerical rank: If k singular values of $\boldsymbol{A}$ are larger than a given number ε, then the ε rank of $\boldsymbol{A}$ is k.

Distance of a matrix of rank n from being a matrix of rank k = σ_{k+1}

Singular Value Decomposition

$$M = \begin{bmatrix} 3 & 0 & 2 \\ 2 & 0 & -2 \\ 0 & 1 & 1 \end{bmatrix}, \quad v = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

```
U, S, Vtranspose = np.linalg.svd(M)
print U
>>> [[-0.95123459  0.23048583 -0.20500982]
     [-0.28736244 -0.90373717  0.31730421]
     [-0.11214087  0.36074286  0.92589903]]

print S
>>> [ 3.72021075  2.87893436  0.93368567]

print Vtranspose
>>> [[-0.9215684  -0.03014369 -0.38704398]
     [-0.38764928  0.1253043   0.91325071]
     [ 0.02096953  0.99166032 -0.12716166]]
```

More Information

Here's a fantastic Python/NumPy tutorial from CS 231N:
http://cs231n.github.io/python-numpy-tutorial/

There's also an IPython notebook containing the above tutorial:
https://github.com/kuleshov/cs228-material/blob/master/tutorials/python/cs228-python-tutorial.ipynb

The rest of the internet!

- NumPy is a very popular package => There's lots written about it!
- Documentation: https://docs.scipy.org/doc/numpy/reference/index.html

Jupyter Notebooks + TensorFlow

Speaking of Jupyter Notebooks, let's look at using them!

Also, since you'll be using TensorFlow on HW2, I'll introduce it now too.

Demo!

TensorFlow

A **very** popular machine learning / *graph computation* framework actively developed and open-sourced by Google in late 2015.

TensorFlow

The most confusing part about using TensorFlow is grappling with its *mental model* and how that translates to code.

Imperative vs. Graph

<u>Imperative</u>: `sum([1.0, 2.0])` => `3.0`

<u>Graph</u>: `d = tf.sum([a, b, c])` => NOTHING! This simply creates a node in the computation graph you're building.

TensorBoard

A TensorFlow visualization dashboard.

```
tensorboard --logdir=<path to log directory>
```

Most commonly used for visualizing training progress.

Demo!

Thanks!

Questions?