Python Introduction and Linear Algebra Review

Boris Ivanovic
CS 231A

April 7, 2017

Previous Iterations of CS 231A used MATLAB

Modern AI uses modern languages => Python is one such language.

- Also doesn’t cost $X,000 per month to use.

For this course we’ll be using Python (specifically Python 2.7).

Outline

Python Review

Linear Algebra Review + Linear Algebra in Python (with NumPy)

Python Review

Python

```
print “Hello, World!”
```

High-level, easy-to-use programming language

You should already be proficient in programming

- Being proficient with Python is a plus, but not strictly necessary

We’ll cover some basics today

Variables

```
a = 6
b = 7
string_var = “Hello, World!”
also_a_string_var = ‘Hello, World!’
c = a+b
print c
>>> 13

print string_var, a, b
>>> Hello, World! 6 7
```

Lists

```
empty_list = list()
also_empty_list = []
zeros_list = [0] * 5
print zeros_list
>>> [0, 0, 0, 0, 0]

empty_list.append(1)
print empty_list
>>> [1]

print len(empty_list)
>>> 1
```

List Indexing

```
list_var = range(10)
print list_var
>>> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print list_var[4]
>>> 4

print list_var[4:7]
>>> [4, 5, 6]

print list_var[0::3] # Empty index means to the beginning/end
>>> [0, 3, 6, 9]
```

List Indexing (cont'd)

```
print list_var
>>> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print list_var[-1]
>>> 9

print list_var[::-1]
>>> [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

print list_var[3:1:-1]
>>> [3, 2]
```

Dictionaries (similar in function to Map in Java)

```
empty_dict = dict()
also_empty_dict = {}
filled_dict = {3: ‘Hello, ’, 4:‘World!’}
print filled_dict[3] + filled_dict[4]
>>> Hello, World!

filled_dict[5] = ‘New String’
print filled_dict
>>> {3: 'Hello, ', 4: 'World!', 5: 'New String'}
```

Dictionaries (cont'd)

```
print filled_dict
>>> {3: 'Hello, ', 4: 'World!', 5: 'New String'}

del filled_dict[3]
print filled_dict
>>> {4: 'World!', 5: 'New String'}

print len(filled_dict)
>>> 2
```

Functions, Lambda Functions

```
def add_numbers(a, b):
   return a + b

print add_numbers(3, 4)
>>> 7

lambda_add_numbers = lambda a, b: a + b
print lambda_add_numbers(3, 4)
>>> 7
```

Loops, List and Dictionary Comprehensions

```
for i in range(10):
    print 'Looping %d' % i
>>> Looping 0
...
>>> Looping 9

filled_list = [a/2 for a in range(10)]
print filled_list
>>> [0, 0, 1, 1, 2, 2, 3, 3, 4, 4]

filled_dict = {a:a**2 for a in range(5)}
print filled_dict
>>> {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Linear Algebra Review + How to do it in Python

Why use Linear Algebra in Computer Vision?

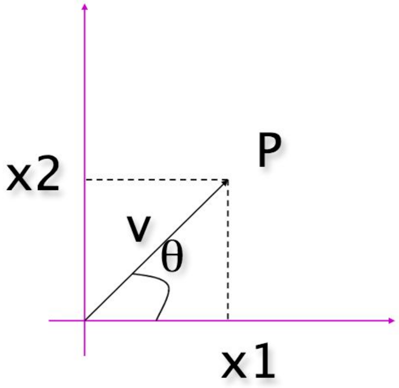
As you’ve seen in lecture, it’s useful to represent many quantities, e.g. 3D points on a scene, 2D points on an image.

Coordinates can be used to perform geometrical transformations and associate 3D points with 2D points (a very common camera operation).

Images are literally matrices filled with numbers (as you will see in HW0).

Vector Review

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



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

If $\|\mathbf{v}\| = 1$, $\mathbf{v}$ Is a UNIT vector

$$\frac{\mathbf{v}}{\|\mathbf{v}\|} = \left(\frac{x_1}{\|\mathbf{v}\|}, \frac{x_2}{\|\mathbf{v}\|} \right)$$ Is a unit vector

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

*Courtesy of last year’s slides.

Vector Review

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

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

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

*Courtesy of last year’s slides.

Matrix Review

$$A_{n\times m} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \dots & \boxed{a_{nm}} \end{bmatrix} \longleftrightarrow$$

Pixel's intensity value

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

A and B must have the same dimensions!

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

*Courtesy of last year's slides.

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]])
```

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
```

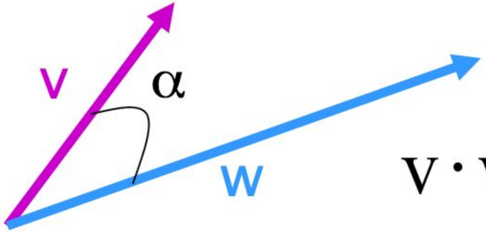
Matrix Indexing

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

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

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

Dot Product



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

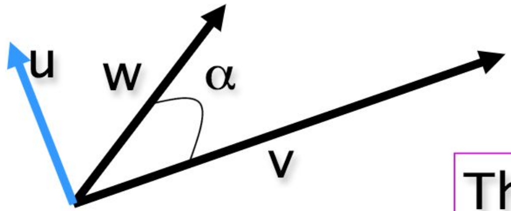
The inner product is a SCALAR!

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

$$\text{if} \quad v \perp w, \quad v \cdot w = ? = 0$$

*Courtesy of last year’s slides.

Cross Product



$$u = v \times w$$

The cross product is a VECTOR!

$$\text{Magnitude:} \|u\| = \|v \times w\| = \|v\| \|w\| \sin \alpha$$

Orientation:

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

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

$$\text{if} \quad v // w\,? \quad \rightarrow u = 0$$

*Courtesy of last year's slides.

Cross Product

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

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

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

$$\mathbf{u} = \mathbf{v} \times \mathbf{w} = (x_1, x_2, x_3) \times (y_1, y_2, y_3)$$

$$= (x_2 y_3 - x_3 y_2)\mathbf{i} + (x_3 y_1 - x_1 y_3)\mathbf{j} + (x_1 y_2 - x_2 y_1)\mathbf{k}$$

*Courtesy of last year’s slides.

Matrix Multiplication

$$A_{n\times m} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix} \mathbf{a}_i \qquad B_{m\times p} = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1p} \\ b_{21} & b_{22} & \dots & b_{2p} \\ \vdots & \vdots & \vdots & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mp} \end{bmatrix}$$

$$\mathbf{b}_j$$

Product:

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

$$c_{ij} = \mathbf{a}_i \cdot \mathbf{b}_j = \sum_{k=1}^{m} a_{ik} b_{kj}$$

A and B must have compatible dimensions!

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

*Courtesy of last year's slides.

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

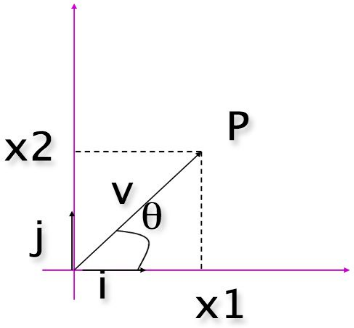
$$M = \begin{bmatrix} 3 & 0 & 2 \\ 2 & 0 & -2 \\ 0 & 1 & 1 \end{bmatrix}, \quad v = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

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

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

Orthonormal Basis

= Orthogonal and Normalized Basis



$$\mathbf{i} = (1,0) \qquad \|\mathbf{i}\| = 1$$

$$\mathbf{j} = (0,1) \qquad \|\mathbf{j}\| = 1$$

$$\mathbf{i} \cdot \mathbf{j} = 0$$

$$\mathbf{v} = (x_1, x_2) \qquad \mathbf{v} = \mathrm{x}_1\mathbf{i} + \mathrm{x}_2\mathbf{j}$$

$$\mathbf{v} \cdot \mathbf{i} = ? \;= (\mathrm{x}_1\mathbf{i} + \mathrm{x}_2\mathbf{j}) \cdot \mathbf{i} = \mathrm{x}_1 1 + \mathrm{x}_2 0 = \mathrm{x}_1$$

$$\mathbf{v} \cdot \mathbf{j} = (\mathrm{x}_1\mathbf{i} + \mathrm{x}_2\mathbf{j}) \cdot \mathbf{j} = \mathrm{x}_1.0 + \mathrm{x}_2.1 = \mathrm{x}_2$$

*Courtesy of last year’s slides.

Transpose

Definition:

$$\mathbf{C}_{m\times n} = \mathbf{A}^T_{n\times m}$$
$$c_{ij} = a_{ji}$$

Identities:

$$(\mathbf{A}+\mathbf{B})^T = \mathbf{A}^T + \mathbf{B}^T$$
$$(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$$

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

*Courtesy of last year's slides.

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 $\mathbf{A}$

$$\det \begin{bmatrix} a_{11} \end{bmatrix} = a_{11}$$

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

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

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

*Courtesy of last year's slides.

Matrix Inverse

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

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

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

If $\det \mathbf{A} = 0$, $\mathbf{A}$ does not have an inverse.

*Courtesy of last year's slides.

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

A eigenvalue λ and eigenvector $\mathbf{u}$ satisfies

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

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

- Multiplying $\mathbf{u}$ by $\mathbf{A}$ scales $\mathbf{u}$ by λ

*Courtesy of last year's slides.

Matrix Eigenvalues and Eigenvectors

Rearranging the previous equation gives the system

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

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

- The eigenvalues are the roots of this determinant which is polynomial in λ.
- Substitute the resulting eigenvalues back into $\mathbf{Au} = \lambda\mathbf{u}$ and solve to obtain the corresponding eigenvector.

*Courtesy of last year's slides.

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]]
```

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

Singular Value Decomposition

Singular values: Non negative square roots of the eigenvalues of $\mathbf{A}^t\mathbf{A}$. Denoted σ_i, $i=1,\ldots,n$

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

$$A = U\,\Sigma\,V^{-1} \qquad \mathrm{U}^{-1}\mathrm{AV} = \Sigma = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \cdot & \\ & & & \sigma_N \end{bmatrix}$$

*Courtesy of last year's slides.

Singular Value Decomposition

Suppose we know the singular values of **A** and we know r are non zero

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

- Rank(**A**) = r.
- Null(**A**) = span$\{\mathbf{v}_{r+1},\dots,\mathbf{v}_n\}$
- Range(**A**)=span$\{\mathbf{u}_1,\dots,\mathbf{u}_r\}$

$$||\mathbf{A}||_F^2 = \sigma_1^2 + \sigma_2^2 + \dots + \sigma_p^2 \qquad ||A||_2 = \sigma_1$$

Numerical rank: If k singular values of A are larger than a given number ε. Then the ε rank of A is k.

Distance of a matrix of rank n from being a matrix of rank $k = \sigma_{k+1}$

*Courtesy of last year's slides.

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]]
```

Recall SVD is the factorization of a matrix into the product of 3 matrices, and is formulated like so:

$$M = U\Sigma V^T$$

More Information

Here's a fantastic Python 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

Office hours!

The rest of the internet!

- Python is a very popular language => There's lots written about it!

Thanks!

Questions