Lecture 15: High Dimensional Data Analysis, Numpy Overview

COMPSCI/MATH 290-04

Chris Tralie, Duke University

3/3/2016

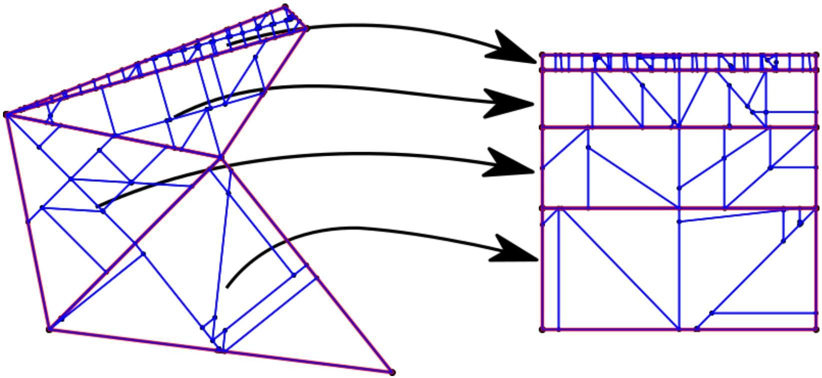
Announcements

- Mini Assignment 3 Out Tomorrow, due next Friday 3/11 11:55PM
- Rank Top 3 Final Project Choices By Tomorrow (Groups of 3-4)
- Dropping Group Assignment 3, Course Grade Schema Change

  Invidiual And Group Programming Assignments 60%
  Final Project 25%
  Midterm Exam 5%
  Class Participation 5%
  Wikipedia Edit 5%
- Midterm Next Thursday 3/10

Table of Contents

3D Surface Equidecomposability Animation

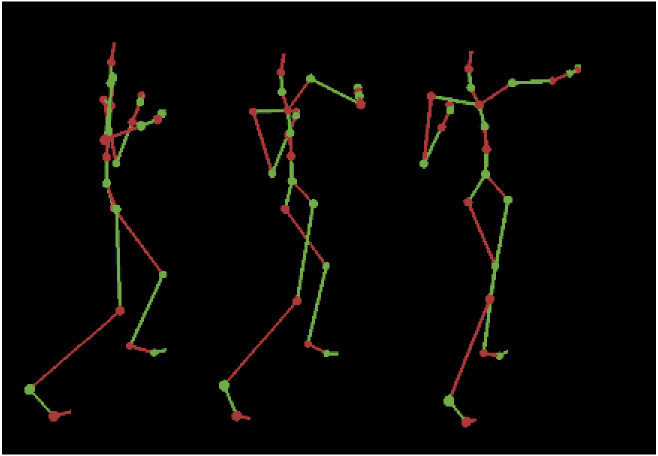


Point Person: Chris Tralie

Ghissi Alterpiece Real Time Rendering

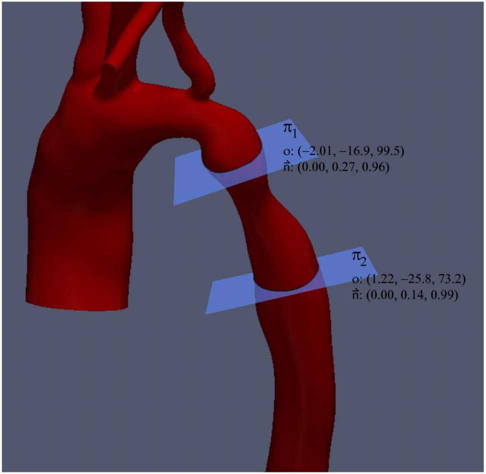
Point Person: Prof Ingrid Daubechies

Motion Capture Javascript Animation



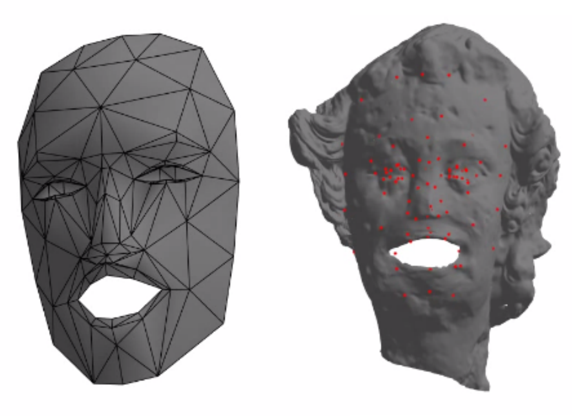
Point People: Chris Tralie / (Prof Ingrid Daubechies?)

Blood Vessel Statistics



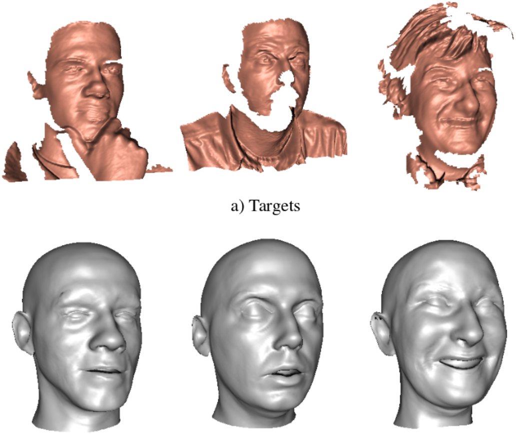
Point People: John Gounley / Prof Amanda Randles

Nasher Museum Talking Heads



Point People: Chris Tralie, Prof Caroline Bruzelius

Face Model Fitting / Morphing



a) Targets

b) Fits

Point People: Jordan Hashemi, Qiang Qiu

Table of Contents

High Dimensional Euclidean Vectors

For *d*-dimensional vectors

$$\vec{a} = (a_1, a_2, \ldots, a_d)$$

$$\vec{b} = (b_1, b_2, \ldots, b_d)$$

High Dimensional Euclidean Vectors

For *d*-dimensional vectors

$$\vec{a} = (a_1, a_2, \ldots, a_d)$$

$$\vec{b} = (b_1, b_2, \ldots, b_d)$$

Vector addition:

$$a \vec{+} b = (a_1 + b_1, a_2 + b_2, \ldots, a_d + b_d)$$

High Dimensional Euclidean Vectors

For *d*-dimensional vectors

$$\vec{a} = (a_1, a_2, \ldots, a_d)$$

$$\vec{b} = (b_1, b_2, \ldots, b_d)$$

Vector addition:

$$a \vec{+} b = (a_1 + b_1, a_2 + b_2, \ldots, a_d + b_d)$$

Vector subtraction:

$$\vec{ab} = (b_1 - a_1, b_2 - a_2, \ldots, b_d - a_d)$$

High Dimensional Euclidean Vectors

Pythagorean Theorem for

$$\vec{a} = (a_1, a_2, \ldots, a_d)$$

$$||\vec{a}|| = \sqrt{a_1^2 + a_2^2 + \ldots + a_d^2}$$

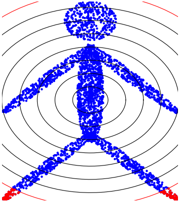
High Dimensional Euclidean Vectors

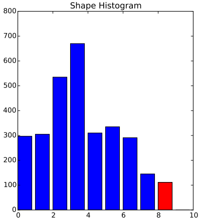
Dot product still holds!

$$\vec{a} \cdot \vec{b} = a_1 b_1 + a_2 b_2 + \ldots + a_d b_d = ||\vec{a}|| ||\vec{b}|| \cos(\theta)$$

Vectors lie on a plane in high dimensions

Histogram Euclidean Distance





For histograms h_1 and h_2

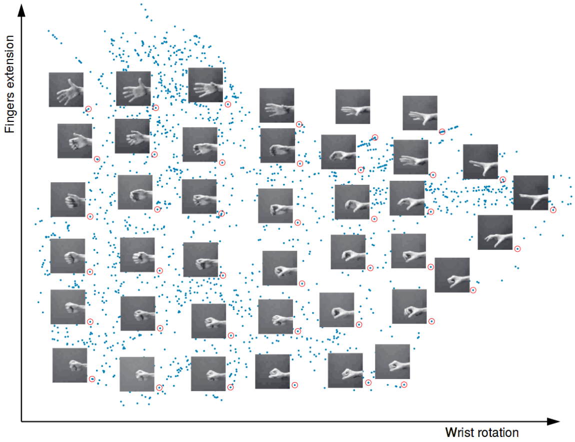
$$d_E(h_1, h_2) = \sqrt{\sum_{i=1}^{N} (h_1[i] - h_2[i])^2}$$

Just thinking of h_1 and h_2 as high dimensional Euclidean vectors! Each histogram bin is a dimension

Histogram Cosine Distance

$$d_C(h_1, h_2) = \cos^{-1}\left(\frac{\vec{h_1} \cdot \vec{h_2}}{||\vec{h_1}||||\vec{h_2}||}\right)$$

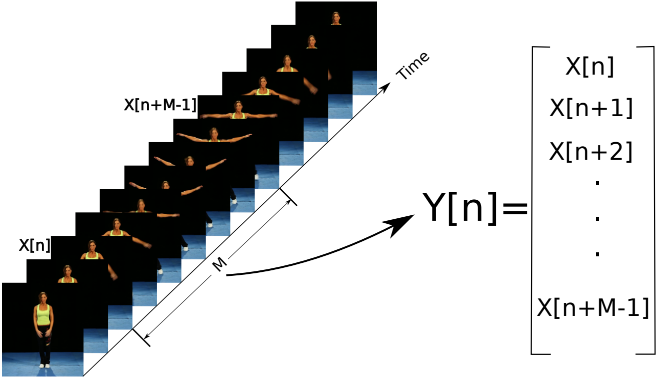
Images Can Be Vectors Too!



One axis per pixel. Above point cloud of images has been flattened to the plane by a nonlinear dimension reduction technique

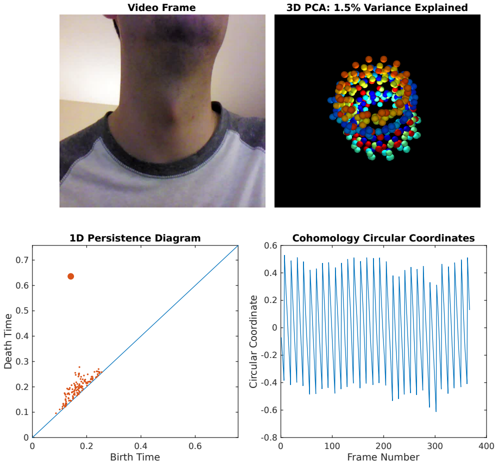
J. B. Tenenbaum, V. de Silva and J. C. Langford

My Work On Video Loops



Tralie 2016

My Work On Video Loops



Tralie 2016

Table of Contents

Evaluation Strategy

Do *leave one out* technique

Use each item as test item in turn, compare to database

- Summarize evaluation statistics over entire database by *averaging them*

Precision / Recall

Rusinkiewiz/Funkhouser 2009

Other Evaluation Metrics

- Average Precision (Area Under Precision/Recall Curve)
- Mean Reciprocal Rank (1/rank of first correct item)
- Median Reciprocal Rank

1 is perfect score

Table of Contents

Python for This Class

- Use Python 2.7
- Switch your editor to use 4 spaces per tab instead of tabs (!!)
- Required Packages: numpy, matplotlib, pyopengl, wxpython
- Optional Packages: scipy (for some extra tasks)
- Helpful Interactive Code Editing: ipython

Python Basics

```
def doSquare(i):
    return i**2

x = []
for i in range(20):
    if i % 2 == 0:
        continue
    x.append(doSquare(i))

#Do a "list comprehension"
x = [doSquare(val) for val in x]
print x
```

Numpy: Array Basics

Numpy = Python + Matlab

```
import numpy as np

np.random.seed(15) #For repeatable results
X = np.round(5*np.random.randn(4, 3)) #Make a random 4x3
     matrix
print X.shape #Tuple that stores dimensions of array
print X, "\n\n"
#Now do some "array slicing"
print X[:, 0], "\n\n" #Access first column
print X[1, :], "\n\n" #Access, second row
print X[3, 2], "\n\n" #Access fourth row, third column
#Unroll into a 1D array row by row
Y = X.flatten()
print Y.shape
print Y, "\n\n"
Y = Y[:, None]
print Y.shape
print Y
```

Numpy: Randomly Subsample

```python
import numpy as np
import matplotlib.pyplot as plt

#Randomly generate 1000 points
np.random.seed(100) #Seed for repeatable results
NPoints = 1000
X = np.random.randn(2, NPoints)
#Randomly subsample 100 points
NSub = 100
Y = X[:, np.random.permutation(NPoints)[0:NSub]]
plt.plot(X[0, :], X[1, :], '.', color='b')
plt.hold(True) #Don't clear the plot when plotting the
    next thing
plt.scatter(Y[0, :], Y[1, :], 20, color='r')
plt.show()
```

Numpy: Boolean Distance Select

```
import numpy as np
import matplotlib.pyplot as plt

#Randomly generate 1000 points
np.random.seed(100) #Seed for repeatable results
NPoints = 1000
X = np.random.randn(2, NPoints)
#Compute distances of points to origin
R = np.sqrt(np.sum(X**2, 0))
#Select points in X with distance greater than 1
#from origin
Y = X[:, R > 1]
#Plot result
plt.plot(Y[0, :], Y[1, :], '.', color='b')
plt.show()
```

Numpy: Boolean Distance Select

```
import numpy as np
import matplotlib.pyplot as plt

#Randomly generate 1000 points
np.random.seed(100) #Seed for repeatable results
NPoints = 1000
X = np.random.randn(2, NPoints)
#Compute distances of points to origin
R = np.sqrt(np.sum(X**2, 0))
#Select points in X with distance greater than 1
#from origin
Y = X[:, R > 1]
#Plot result
plt.plot(Y[0, :], Y[1, :], '.', color='b')
plt.show()
```

Numpy: Broadcasting, Rotate Ellipse

```
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(404)
X = np.random.randn(2, 300)
#Scale X by "broadcasting"
X = np.array([[5], [1]])*X
#Setup a rotation matrix
[C, S] = [np.cos(np.pi/4), np.sin(np.pi/4)]
R = np.array([[C, -S], [S, C]])
#Multiply points on the left by the rotation matrix
Y = R.dot(X)
#Set axes equal scale
plt.axes().set_aspect('equal', 'datalim')
plt.plot(Y[0, :], Y[1, :], '.')
plt.show()
```

Numpy: Broadcasting, Sphere Normalization

```
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(404)
X = np.random.randn(2, 300)
#Normalize each column
XNorm = np.sqrt(np.sum(X**2, 0))
#Broadcast 1/XNorm to each row
Y = X/XNorm
plt.plot(Y[0, :], Y[1, :], '.')
plt.show()
```

Numpy: More Broadcasting

```
import numpy as np
import matplotlib.pyplot as plt
X = np.arange(4)
Y = np.arange(6)
Z = X[:, None] + Y[None, :]
print Z
```

Numpy: PCA Implementation

```
import numpy as np
import matplotlib.pyplot as plt
#Make a sinusoid point cloud
t = np.linspace(0, 2*np.pi, 100)
X = np.zeros((2, len(t)))
X[0, :] = t
X[1, :] = np.sin(t)
#Mean-center
X = X-np.mean(X, 1)[:, None]
#Do PCA
D = X.dot(X.T) #X*X Transpose
[eigs, V] = np.linalg.eig(D) #Eigenvectors in columns
eigs = np.sqrt(eigs/X.shape[1]) #Make average dot
    product length
#Scale columns by eigenvectors
V = V*eigs[None, :]
plt.plot(X[0, :], X[1, :], '.'); plt.hold(True)
#First eigvec is in first column, second in second
plt.arrow(0, 0, V[0, 0], V[1, 0], ec = 'r')
plt.arrow(0, 0, V[0, 1], V[1, 1], ec = 'g')
plt.axes().set_aspect('equal', 'datalim'); plt.show()
```

Squared Euclidean Distances in Matrix Form

Notice that

$$||\vec{a}-\vec{b}||^2 = (\vec{a}-\vec{b})\cdot(\vec{a}-\vec{b})$$

$$||\vec{a}-\vec{b}||^2 = \vec{a}\cdot\vec{a} + \vec{b}\cdot\vec{b} - 2\vec{a}\cdot\vec{b}$$

Squared Euclidean Distances in Matrix Form

Notice that

$$||\vec{a} - \vec{b}||^2 = (\vec{a} - \vec{b}) \cdot (\vec{a} - \vec{b})$$

$$||\vec{a} - \vec{b}||^2 = \vec{a} \cdot \vec{a} + \vec{b} \cdot \vec{b} - 2\vec{a} \cdot \vec{b}$$

Given points clouds X and Y expressed as $2 \times M$ and $2 \times N$ matrices, respectively, write code to compute an $M \times N$ matrix D so that

$$D[i, j] = ||X[:, i] - Y[:, j]||^2$$

Without using any for loops! Can use for ranking with Euclidean distance or D2 shape histograms, for example

Brute Force Nearest Neighbors

```
import numpy as np
import matplotlib.pyplot as plt
t = np.linspace(0, 2*np.pi, 100)
X = np.zeros((2, len(t)))
X[0, :] = t
X[1, :] = np.cos(t)
Y = np.zeros((2, len(t)))
Y[0, :] = t
Y[1, :] = np.sin(t**1.2)
##FILL THIS IN TO COMPUTE DISTANCE MATRIX D
idx = np.argmin(D, 1) #Find index of closest point in Y
    to point in X
plt.plot(X[0, :], X[1, :], '.')
plt.hold(True)
plt.plot(Y[0, :], Y[1, :], '.', color = 'red')
for i in range(len(idx)):
    plt.plot([X[0, i], Y[0, idx[i]]], [X[1, i], Y[1, idx
        [i]]], 'b')
plt.axes().set_aspect('equal', 'datalim'); plt.show()
```