NUS-ISS
Intelligent Sensing and Sense Making

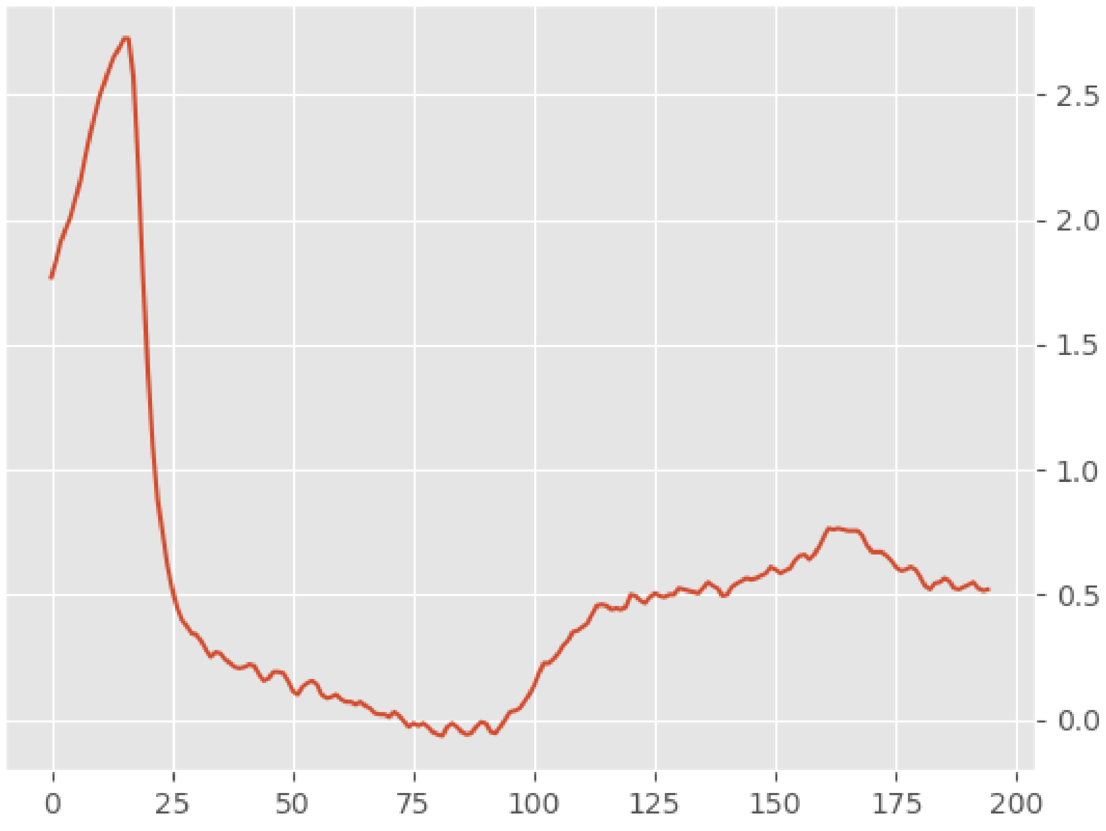
Module 3 - Workshop on sensor signal processing

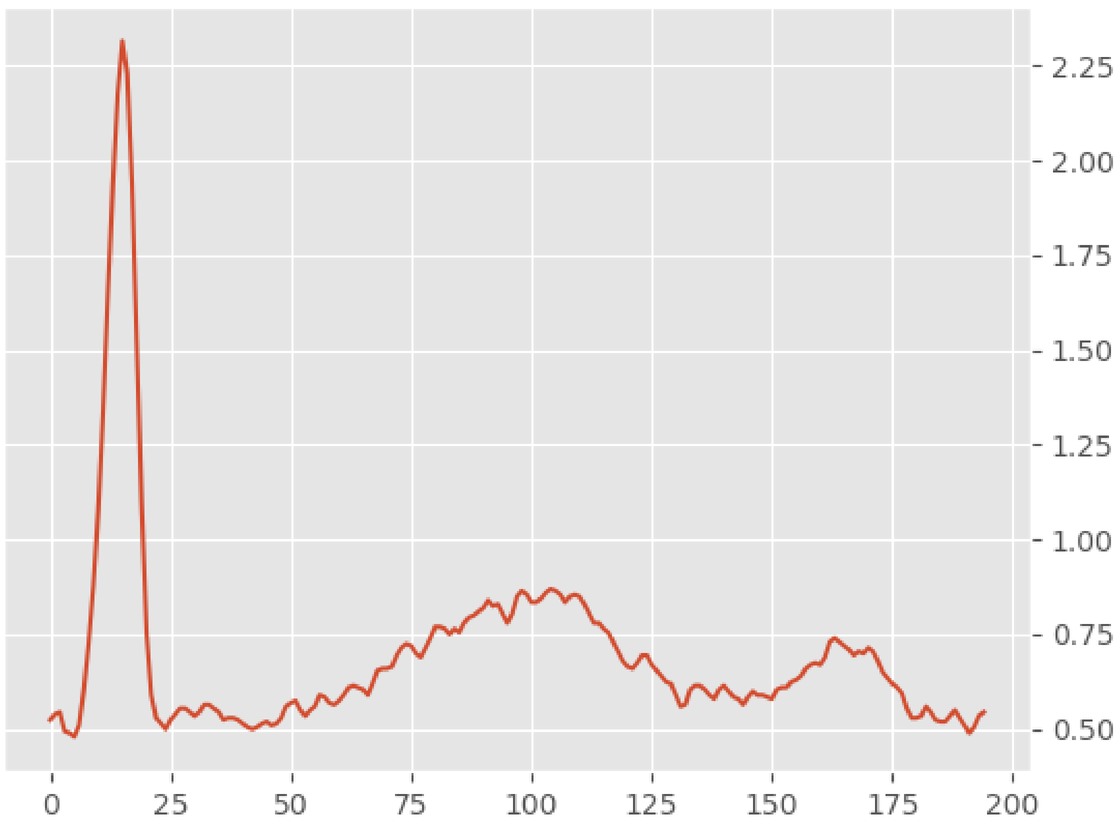
by Nicholas Ho

Dynamic Time Warping

Problem

- How similar are these two signals?
- In which manners are they similar?

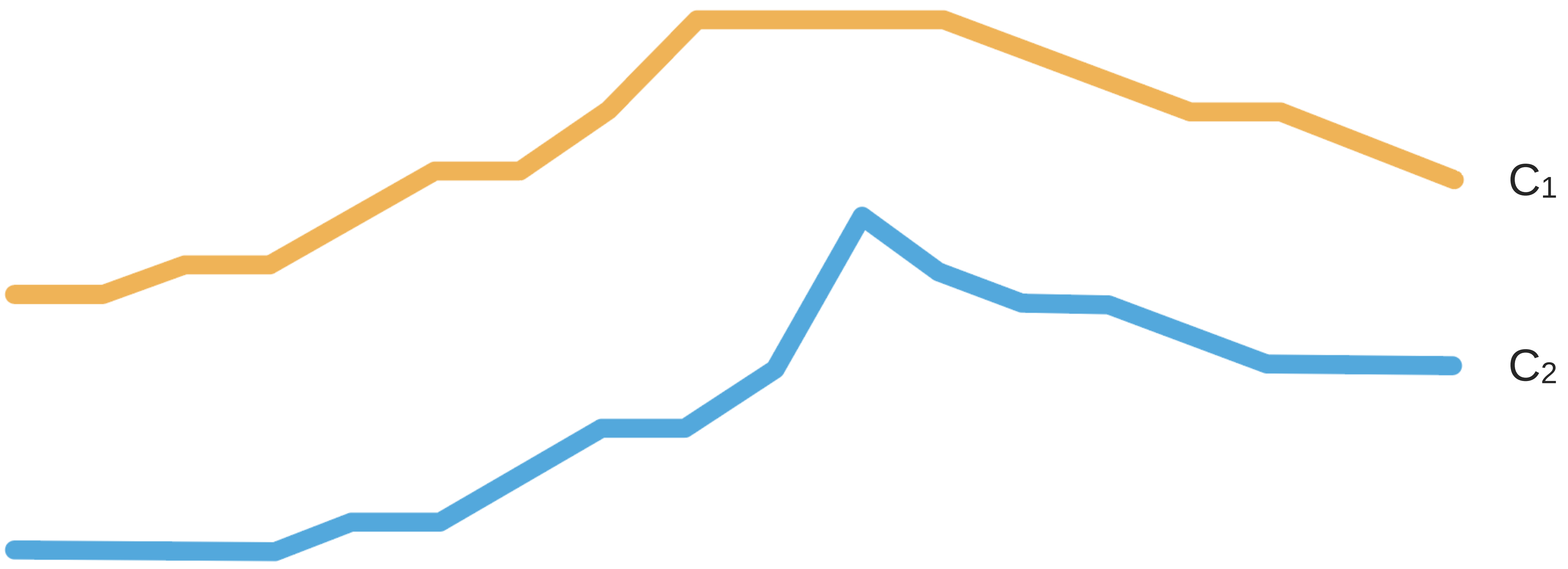




Similarity

How similar are these two signals .....?

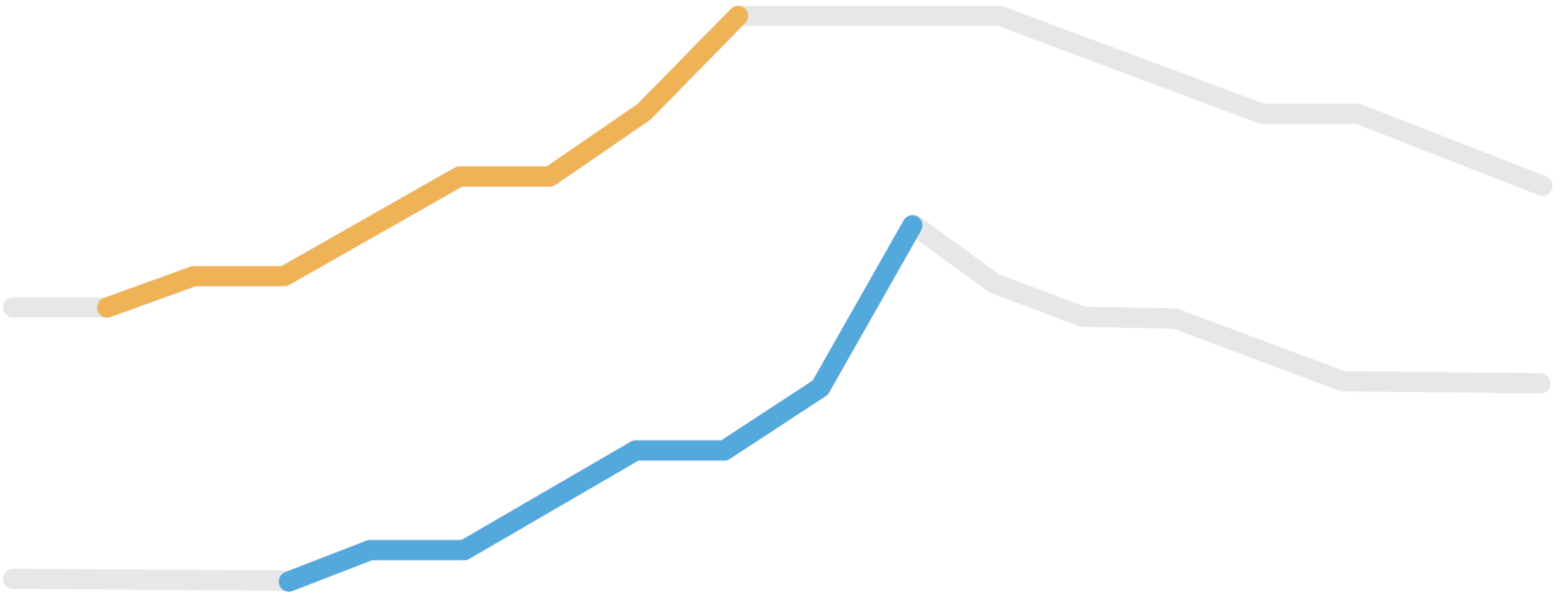
- In what ways are these two signals similar to each other?



Similarity

How similar are these two signals .....?

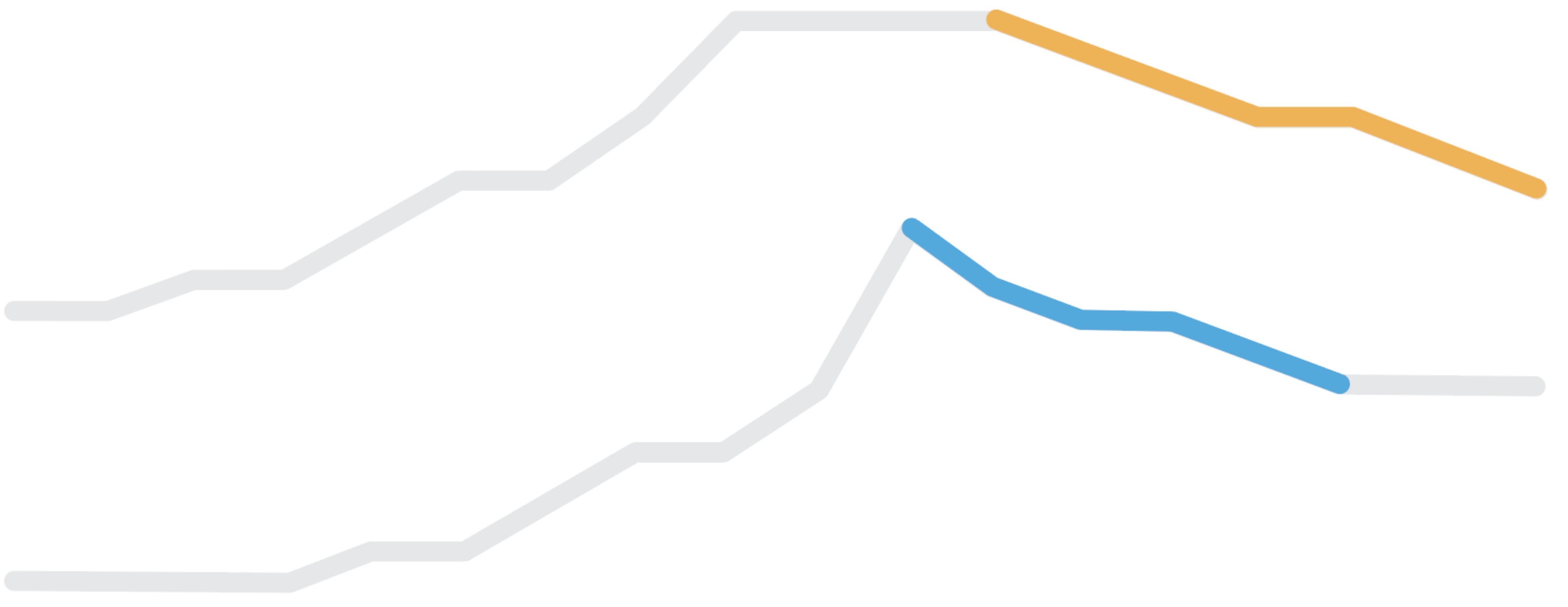
- In what ways are these two signals similar to each other?



Similarity

How similar are these two signals .....?

- In what ways are these two signals similar to each other?

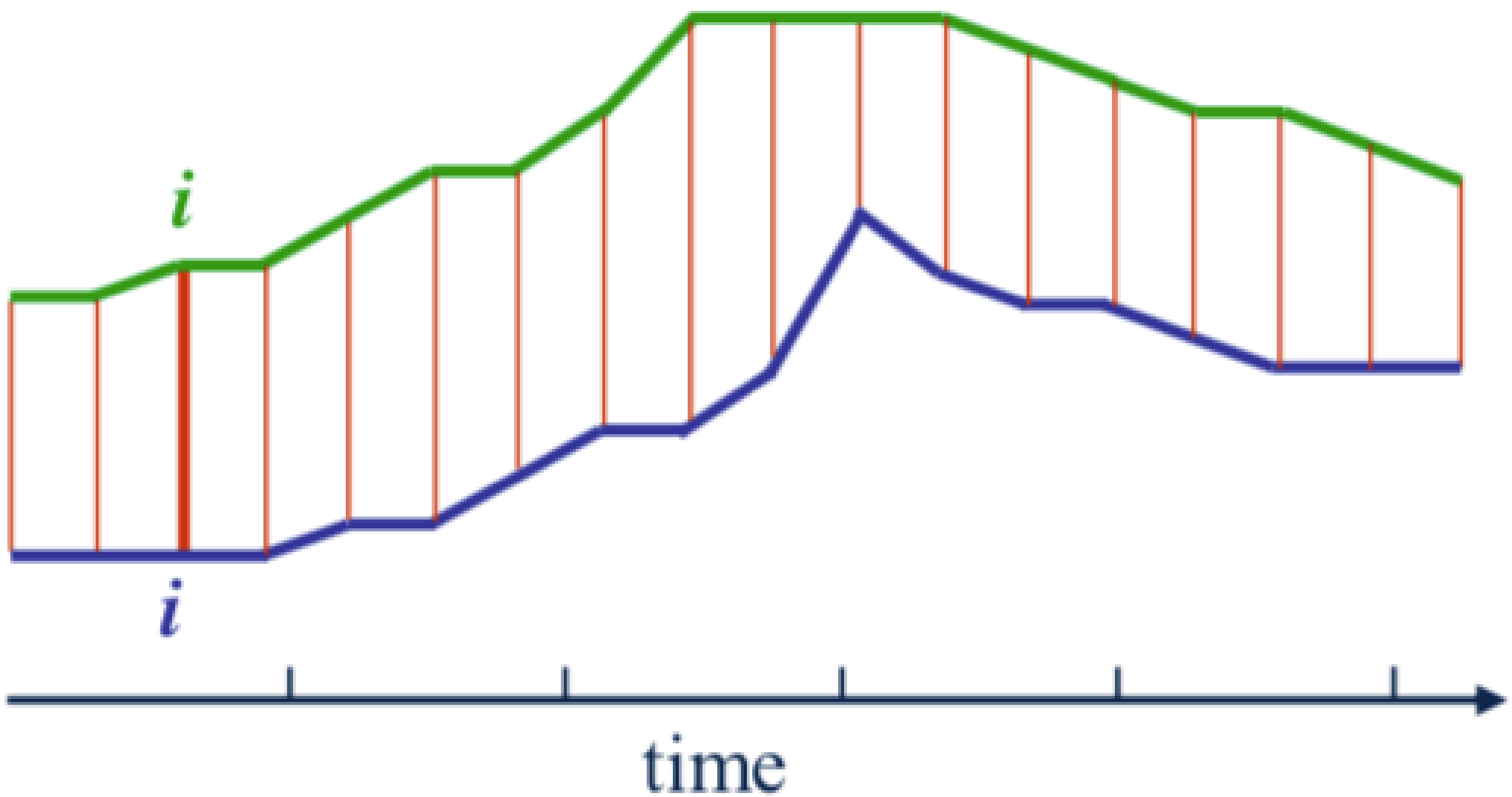


Can we try ...

Euclidean, Manhattan .....?

- We can measure the similarity of two signals by calculating the distance between the i-th point on one signal and the i-th point on another signal

- Simple concept, but could not capture the similarity in shape



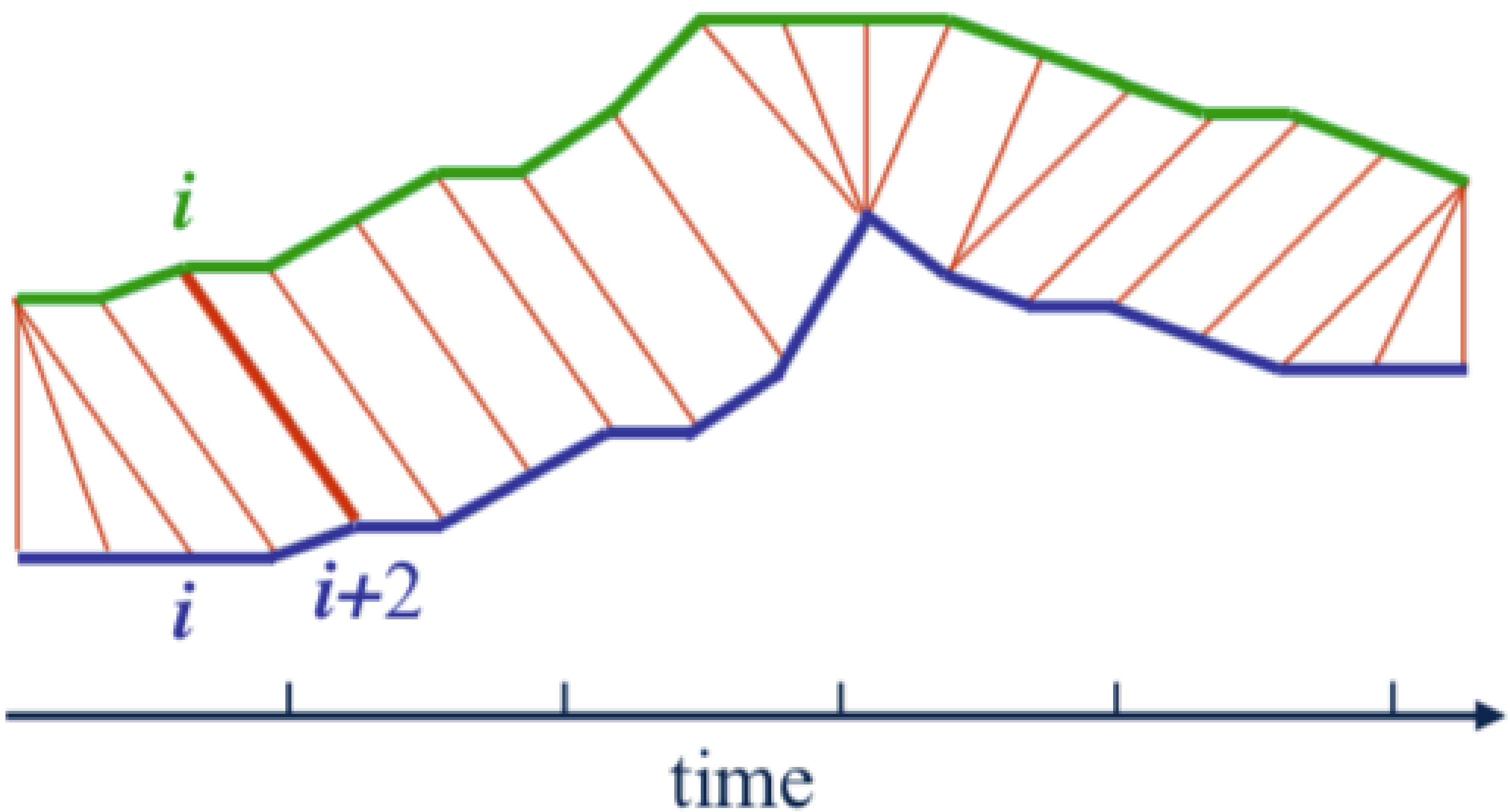
Source: "Dynamic Time Warping Algorithm", by Elena Tsiporkova

How about ...

non-linear alignment .....?

- Elastic alignment between points of two signals produces a better, more intuitive similarity measure

- Allow similar shapes to match even if they are out of phase



Source: "Dynamic Time Warping Algorithm", by Elena Tsiporkova

Distance

Another term to say 'similiarity'

- Consider two distinct signals

$$\mathbf{x} = [x_1, x_2, \ldots, x_i, \ldots x_m]$$
$$\mathbf{y} = [y_1, y_2, \ldots, y_j, \ldots y_n]$$

- The distance between the two signals is defined as

$$d_s(\mathbf{x}, \mathbf{y})$$

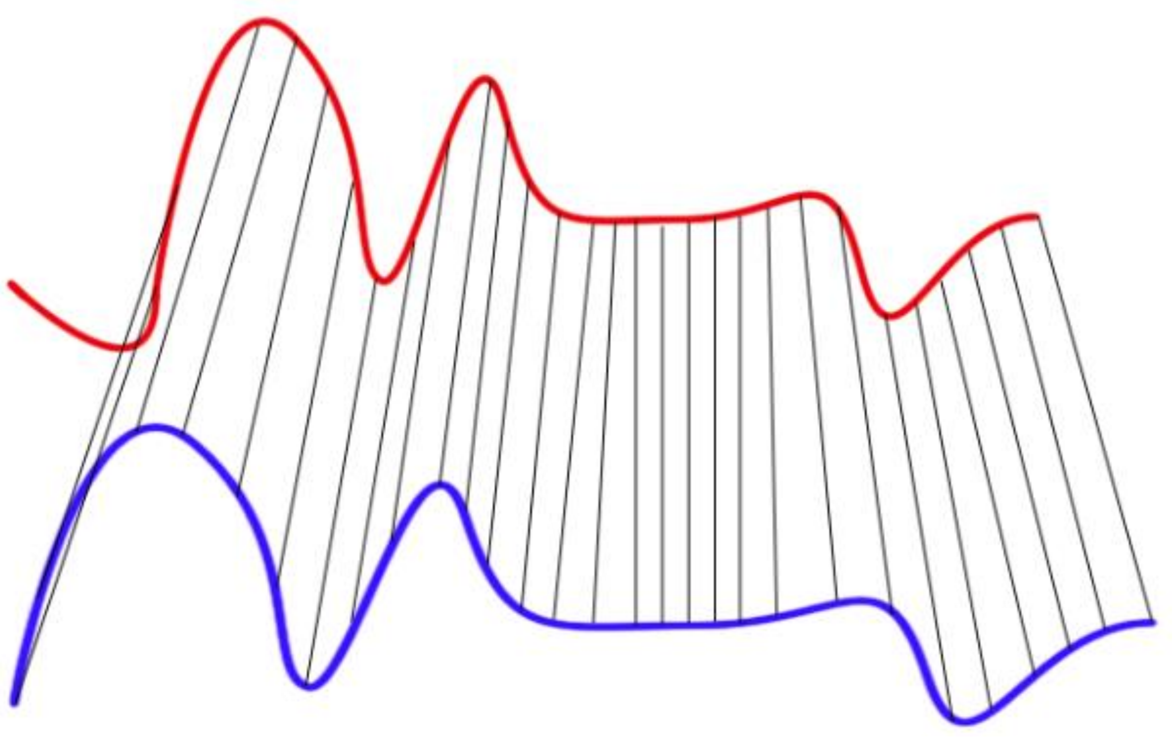
- Euclidean distance between two signals:

$$d_s(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_i (x_i - y_i)^2}$$

- **Problem: two signals must be of same length!**

Dynamic Time Warping
DTW

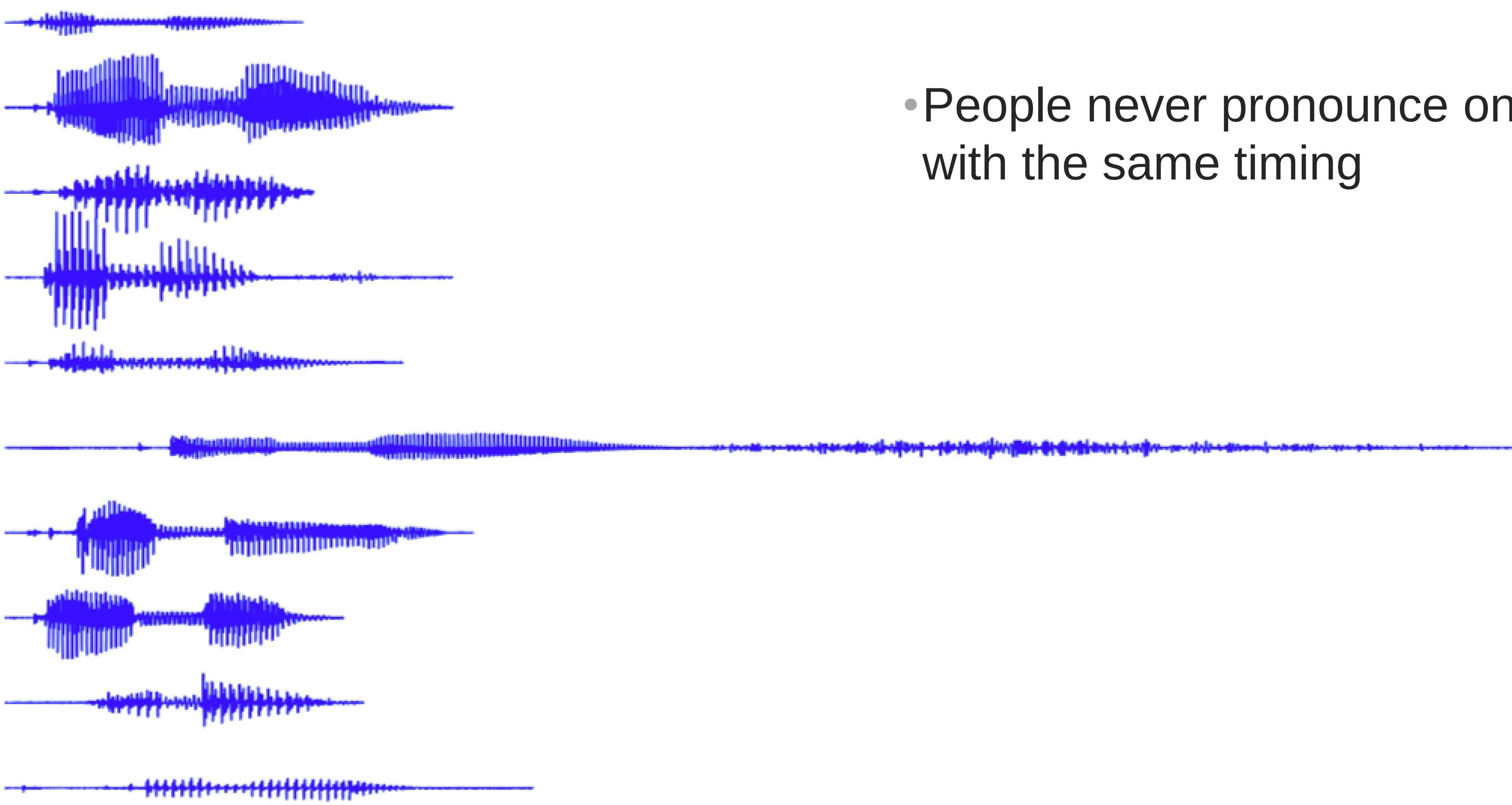
- An algorithm to measure similarity between two temporal sequences (signal), which may vary in speed
- DTW calculates an optimal match between two given sequences
- Sequences are warped along time dimension to determine similarity independent of variations in time
- DTW produces warping path, which enables alignment between two signals



Source: https://th.wikipedia.org/wiki/Dynamic_time_warping#/media/File:Euclidean_vs_DTW.jpg

Dynamic Time Warping
Usage

"timing"

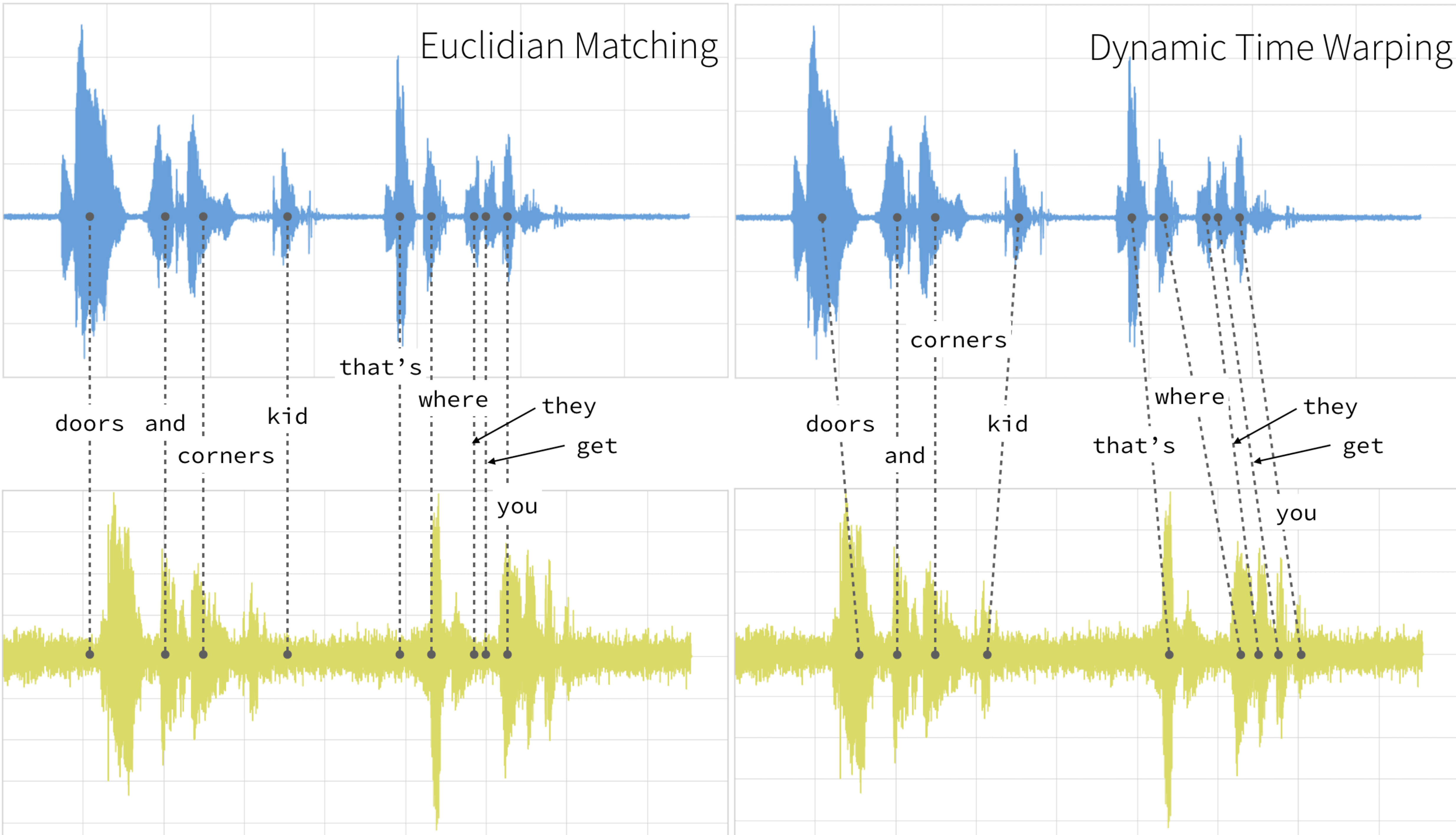


Source: "Speech Recognition - Intro and DTW", by Jan Černocky

- Commonly used in speech recognition
- Individual never pronounces one word twice in exact way
- People never pronounce one word with the same timing

Dynamic Time Warping

Vs Euclidean (Comparison of Audio Clips Example)

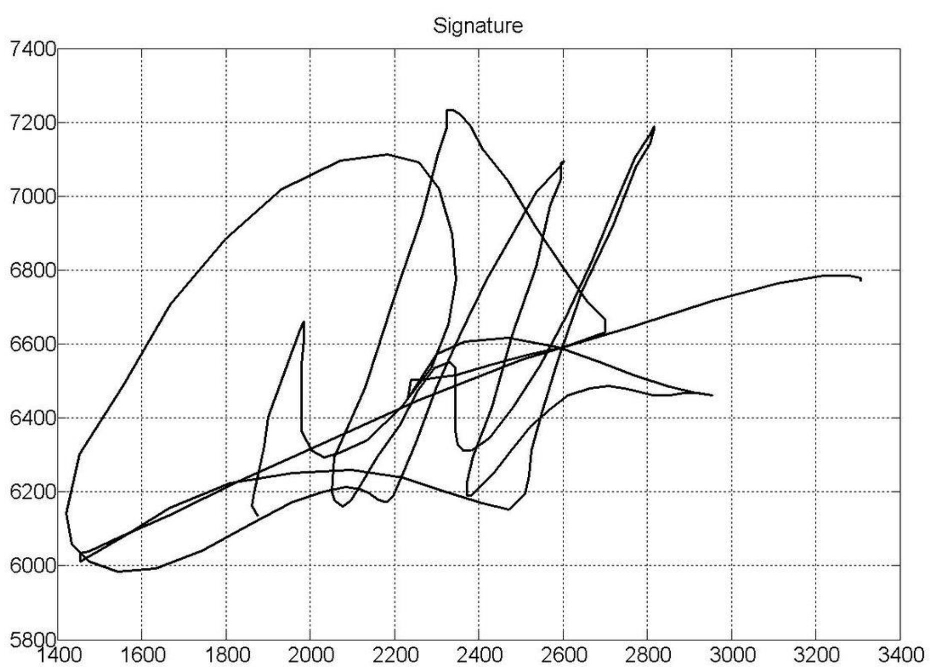


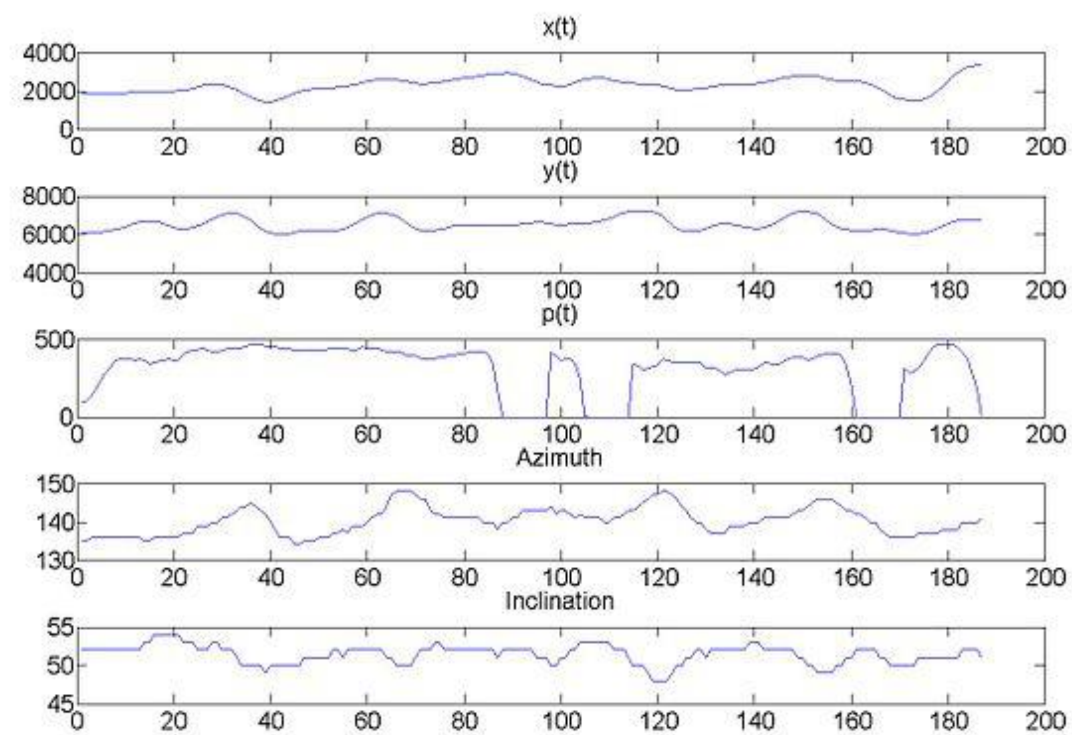
Dynamic Time Warping

Dynamic signature recognition

- Users sign their signature on digital tablet
- Dynamic information captured:
  - x position
  - y position
  - pressure
  - azimuth
  - inclination

- Use DTW to check / match signature

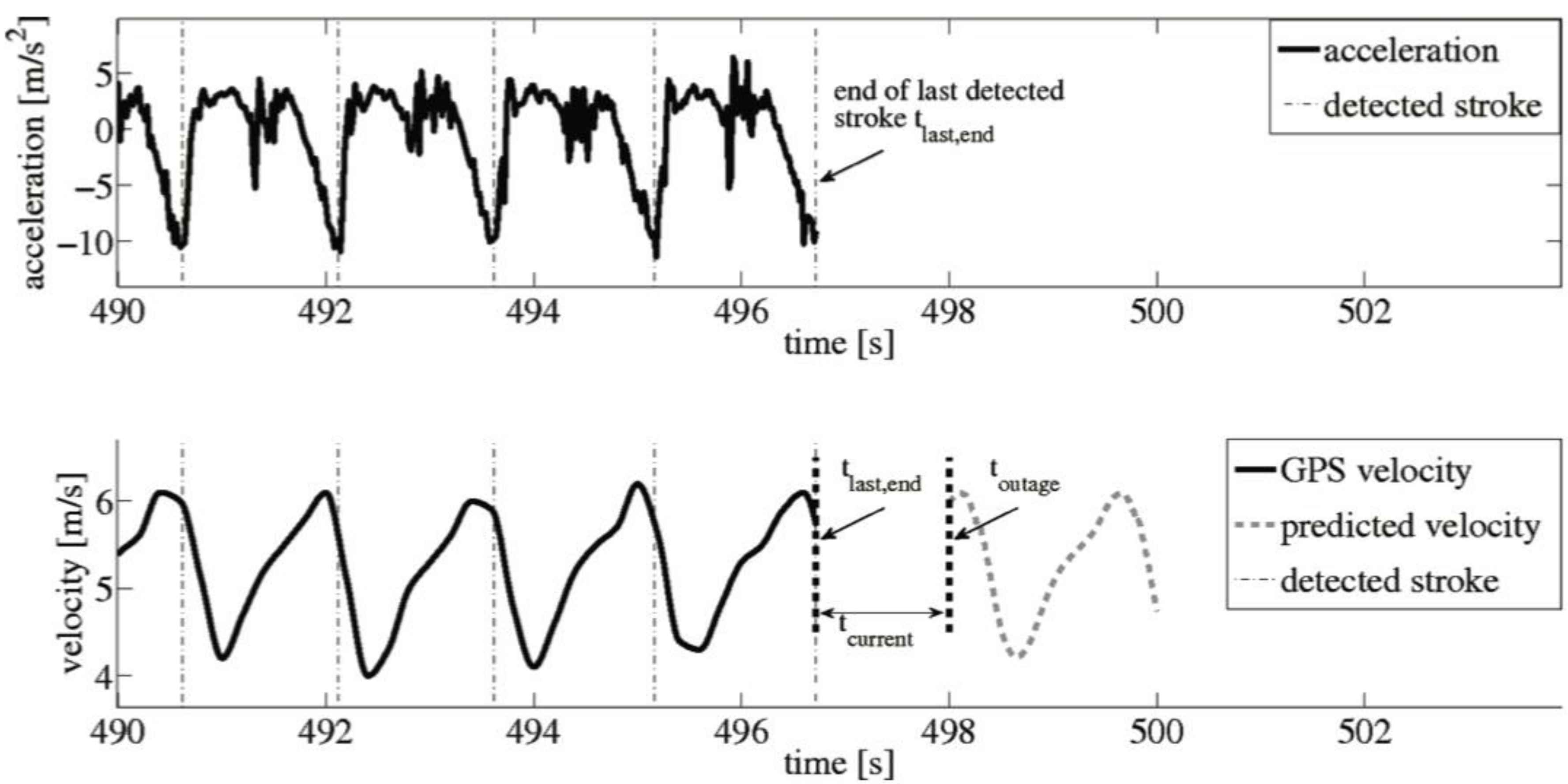




Source: "Speech Recognition - Intro and DTW", by Jan Černocky

Dynamic Time Warping
Stroke detection (rowing)

- Use DTW to detect stroke (used in rowing competitions)
- With strokes detected, predict boat's movement and position when sensor transmission lost

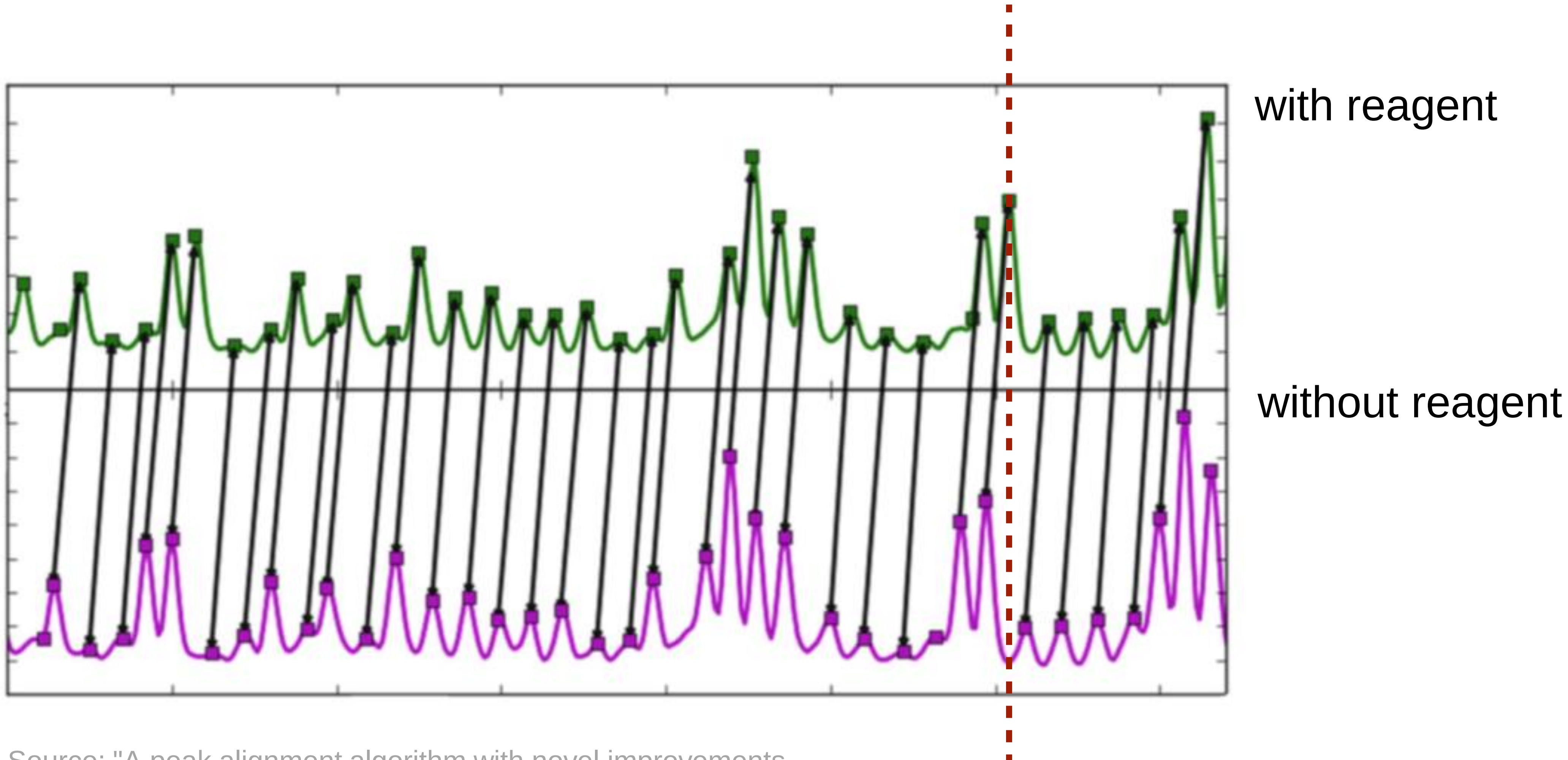


Source: "Movement prediction in rowing using a dynamic time warping base stroke detection", by Groh et al.

Dynamic Time Warping

Peak alignment in DNA sequencing

- Use DTW to align peaks in electropherogram (a plot generated by DNA sequencer)
- Accurate alignment gives better interpretation (e.g. better RNA secondary structure prediction)



Source: "A peak alignment algorithm with novel improvements in application to electropherogram analysis", by Karabiber

Dynamic Time Warping

Overview of algorithm

n x m matrix covers the window boundaries

orange curve = correct wrapping path

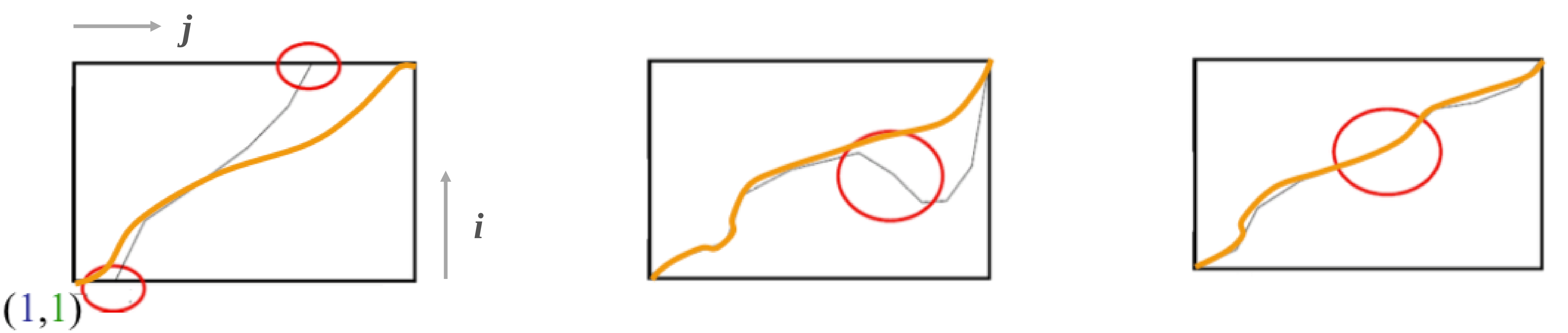
- Start by constructing $\overset{\text{row}}{n}$ x $\overset{\text{clm}}{m}$ matrix *D*, in which

$$D_{i,j} = d_s(y_i, x_j)$$

- where

$$d_s(y_i, x_j) = (y_i - x_j)^2$$

- Create a warping path *w* that maps points between x and y, the path w must satisfy the following rules:
  1. Boundary conditions
  2. Monotonicity
  3. Continuity



Source: "Dynamic time warping algorithm", by Elena Tsiporkova

Dynamic Time Warping

Overview of algorithm

<u>1st Rule:</u> start point from bottom left and end point at top right

<u>2nd Rule:</u> Cannot move back in direction

<u>3rd Rule:</u> cannot have a break in between! Must be continuous

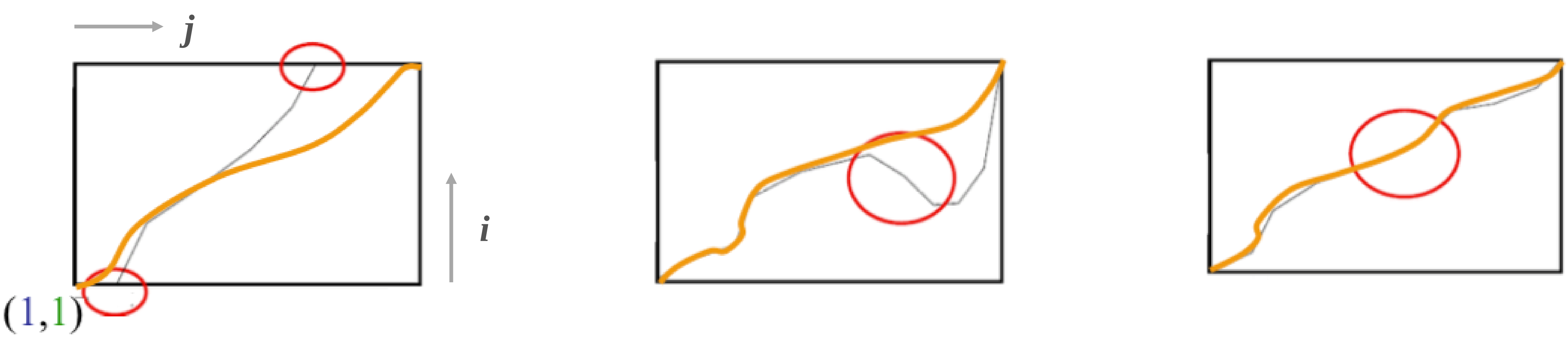
- Start by constructing $\overset{\text{row}}{n}$ x $\overset{\text{clm}}{m}$ matrix *D*, in which

$$D_{i,j} = d_s(y_i, x_j)$$

- where

$$d_s(y_i, x_j) = (y_i - x_j)^2$$

- Create a warping path *w* that maps points between x and y, the path w must satisfy the following rules:
  1. Boundary conditions
  2. Monotonicity
  3. Continuity



Source: "Dynamic time warping algorithm", by Elena Tsiporkova

Dynamic Time Warping

Overview of algorithm

Codes to import libraries and to perform some setup

- DTW algorithm consists of mainly 3 parts:
  1. Compute distance matrix
  2. Compute accumulated cost matrix
  3. Search the optimal path

- To start the code, import the necessary libraries, and setup a bit

```
> import numpy as np
> import matplotlib.pyplot as plt
> import pandas as pd

> plt.style.use('ggplot')
> plt.rcParams['ytick.right']      = True
> plt.rcParams['ytick.labelright'] = True
> plt.rcParams['ytick.left']       = False
> plt.rcParams['ytick.labelleft'] = False
```

Dynamic Time Warping

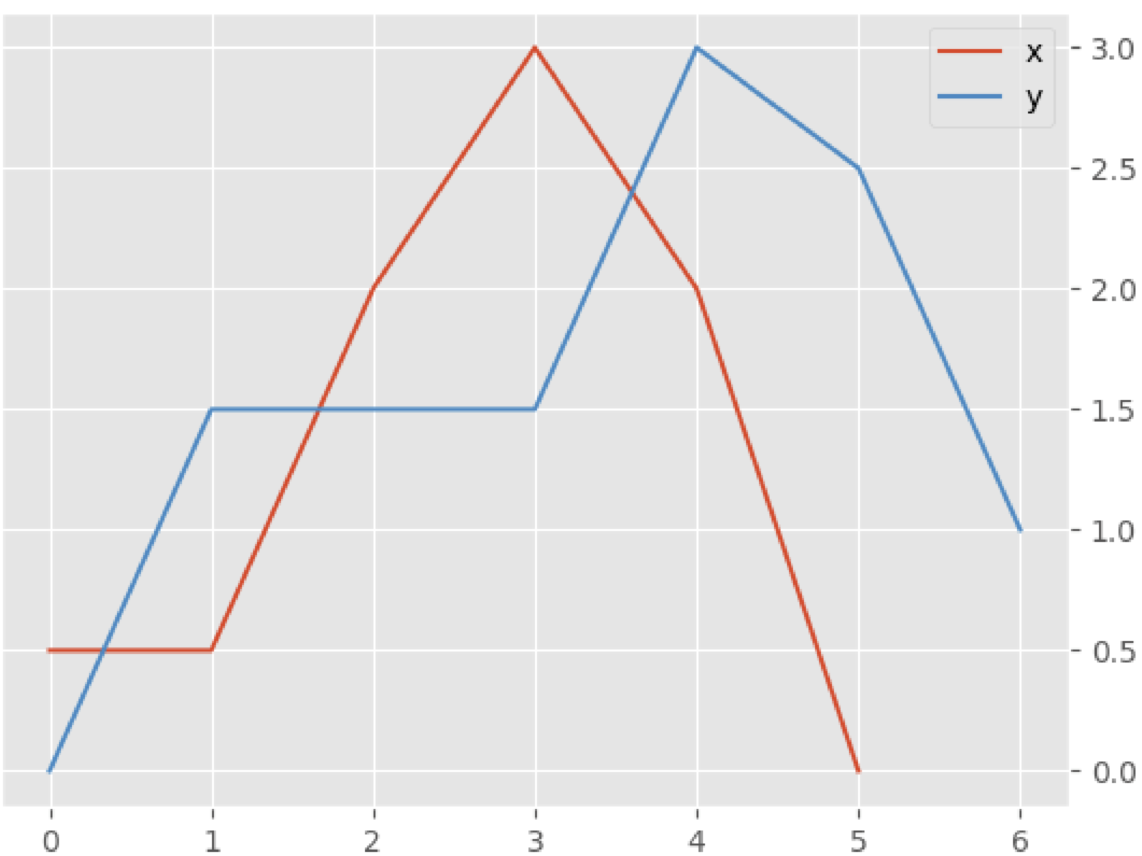
1. Compute distance matrix

- Define two simple short signals:

```
> x = np.array([0.5,0.5,2.0,3.0,2.0,0.0])
> y = np.array([0.0,1.5,1.5,1.5,3.0,2.5,1.0])
```

- Plot the two signals

```
> plt.figure()
> plt.plot(x,
           color="C0",
           label='x')
> plt.plot(y,
           color="C1",
           label='y')
> plt.legend()
```



Dynamic Time Warping

1. Compute distance matrix

applying maths formula to calculate the distance matrix in DTW

dists

```
[[0.25, 0.25, 4.  , 9.  , 4.  , 0.  ],
 [1.  , 1.  , 0.25, 2.25, 0.25, 2.25],
 [1.  , 1.  , 0.25, 2.25, 0.25, 2.25],
 [1.  , 1.  , 0.25, 2.25, 0.25, 2.25],
 [6.25, 6.25, 1.  , 0.  , 1.  , 9.  ],
 [4.  , 4.  , 0.25, 0.25, 0.25, 6.25],
 [0.25, 0.25, 1.  , 4.  , 1.  , 1.  ]]
```

- Compute the distance matrix is straightforward, since the matrix is defined as

$$D_{i,j} = d_s(y_i, x_j)$$

- and

$$d_s(y_i, x_j) = (y_i - x_j)^2$$

- The corresponding code

```
> dists = np.zeros((len(y),len(x)))

> for i in range(len(y)):
      for j in range(len(x)):
          dists[i,j]  = (y[i]-x[j])**2
```

Dynamic Time Warping

1. Compute distance matrix

<u>Note:</u> Output from these codes is inverted

i.e. top left cell = D[0,0]; bottom left should be D[0,0] instead

dists

```
[[0.25, 0.25, 4.  , 9.  , 4.  , 0.  ],
 [1.  , 1.  , 0.25, 2.25, 0.25, 2.25],
 [1.  , 1.  , 0.25, 2.25, 0.25, 2.25],
 [1.  , 1.  , 0.25, 2.25, 0.25, 2.25],
 [6.25, 6.25, 1.  , 0.  , 1.  , 9.  ],
 [4.  , 4.  , 0.25, 0.25, 0.25, 6.25],
 [0.25, 0.25, 1.  , 4.  , 1.  , 1.  ]]
```

- Compute the distance matrix is straightforward, since the matrix is defined as

$$D_{i,j} = d_s(y_i, x_j)$$

- and

$$d_s(y_i, x_j) = (y_i - x_j)^2$$

- The corresponding code

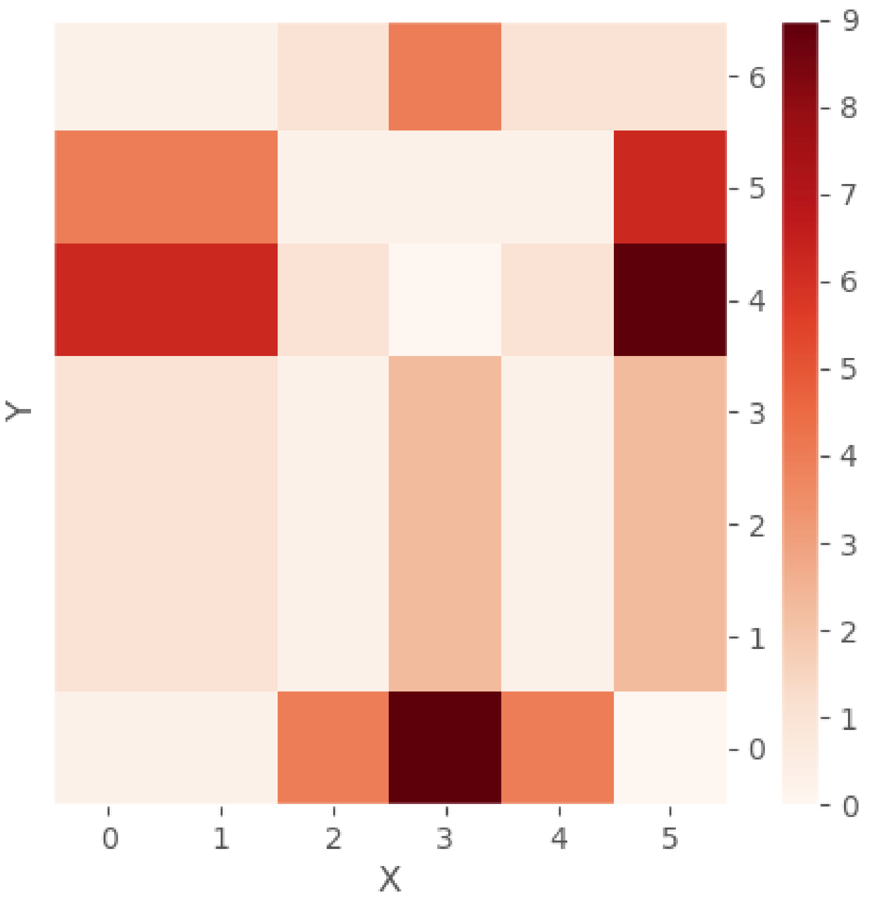
```
> dists = np.zeros((len(y),len(x)))

> for i in range(len(y)):
      for j in range(len(x)):
          dists[i,j]  = (y[i]-x[j])**2
```

Dynamic Time Warping
1. Compute distance matrix

- Create a function to do a plot on the distance matrix

Inverted because viridis starts from top left

```
> def pltDistances(dists,xlab="X",ylab="Y",clrmap="viridis"):
      imgplt  = plt.figure()
      plt.imshow(dists,
                 interpolation='nearest',
                 cmap=clrmap)

      plt.gca().invert_yaxis()
      plt.xlabel(xlab)
      plt.ylabel(ylab)
      plt.grid()
      plt.colorbar()

      return imgplt

> pltDistances(dists,clrmap='Reds')
```

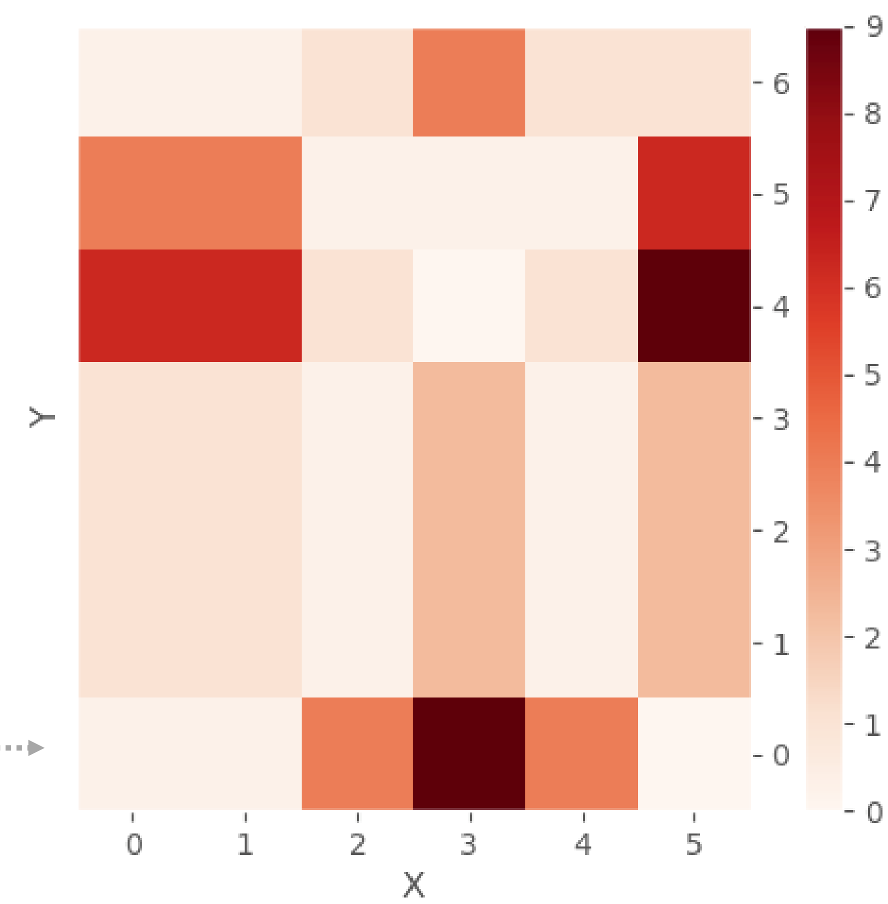


Dynamic Time Warping

1. Compute distance matrix

- Do take note, in the plot, the y axis is inverted
- Thus, first row of the matrix corresponds to the last row in the figure

```
[[0.25, 0.25, 4.  , 9.  , 4.  , 0.  ],
 [1.  , 1.  , 0.25, 2.25, 0.25, 2.25],
 [1.  , 1.  , 0.25, 2.25, 0.25, 2.25],
 [1.  , 1.  , 0.25, 2.25, 0.25, 2.25],
 [6.25, 6.25, 1.  , 0.  , 1.  , 9.  ],
 [4.  , 4.  , 0.25, 0.25, 0.25, 6.25],
 [0.25, 0.25, 1.  , 4.  , 1.  , 1.  ]]
```

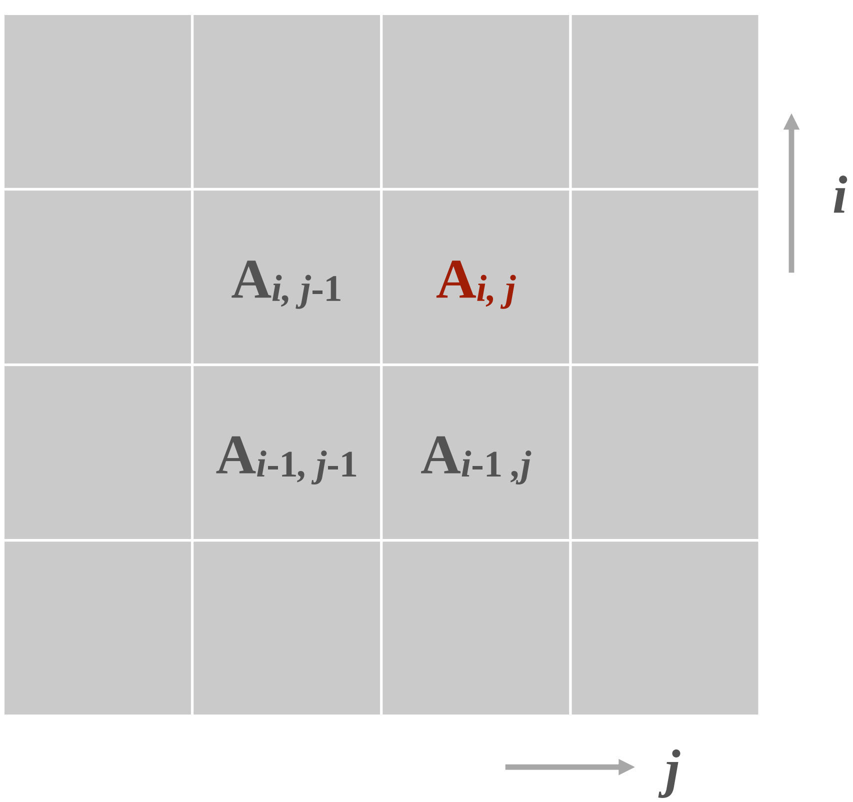


Dynamic Time Warping

2. Compute accumulated cost matrix

Cost matrix important to determine wrapping path!

$\mathbf{A}_{i,j}$ equals to $\mathbf{D}_{i,j}$ plus either $\mathbf{A}_{i-1,j-1}$, $\mathbf{A}_{i,j-1}$ or $\mathbf{A}_{i-1,j}$, whichever has the lowest value



- The accumulated cost matrix is defined

$$A_{i,j} = D_{i,j} + min(A_{i-1,j}, A_{i,j-1}, A_{i-1,j-1})$$

- When *i* and *j* equals to 0

$$A_{0,0} = D_{0,0}$$

- When *i* equals to 0 (first row)

$$A_{0,j} = D_{0,j} + A_{0,j-1}$$

- When *j* equals to 0 (first column)

$$A_{i,0} = D_{i,0} + A_{i-1,0}$$

Dynamic Time Warping

2. Compute accumulated cost matrix

- When *i* equals to 0 (first row)

$$A_{0,j} = D_{0,j} + A_{0,j-1}$$

- When *j* equals to 0 (first column)

$$A_{i,0} = D_{i,0} + A_{i-1,0}$$

j

D

0.25	0.25	1	4	1	1
4	4	0.25	0.25	0.25	6.25
6.25	6.25	1	0	1	9
1	1	0.25	2.25	0.25	2.25
1	1	0.25	2.25	0.25	2.25
1	1	0.25	2.25	0.25	2.25
0.25	**0.25**	4	9	4	0

i

A

0.25					

Dynamic Time Warping

2. Compute accumulated cost matrix

- When i equals to 0 (first row)

$$A_{0,j} = D_{0,j} + A_{0,j-1}$$

- When j equals to 0 (first column)

$$A_{i,0} = D_{i,0} + A_{i-1,0}$$

j

D

0.25	0.25	1	4	1	1
4	4	0.25	0.25	0.25	6.25
6.25	6.25	1	0	1	9
1	1	0.25	2.25	0.25	2.25
1	1	0.25	2.25	0.25	2.25
1	1	0.25	2.25	0.25	2.25
0.25	0.25	4	9	**4**	0

i

A

9.5					
3.25					
2.25					
1.25					
0.25	0.5	4.5	**13.5**		

Dynamic Time Warping

2. Compute accumulated cost matrix

- Else

$$A_{i,j} = D_{i,j} + min(A_{i-1,j}, A_{i,j-1}, A_{i-1,j-1})$$

j

0.25	0.25	1	4	1	1
4	4	0.25	0.25	0.25	6.25
6.25	6.25	1	0	1	9
1	1	0.25	2.25	0.25	2.25
1	1	0.25	2.25	0.25	2.25
1	**1**	0.25	2.25	0.25	2.25
0.25	0.25	4	9	4	0

D

i

13.75					
13.5					
9.5					
3.25					
2.25					
1.25					
0.25	0.5	4.5	13.5	17.5	17.5

A

Dynamic Time Warping

2. Compute accumulated cost matrix

- Else

$$A_{i,j} = D_{i,j} + min(A_{i-1,j}, A_{i,j-1}, A_{i-1,j-1})$$

$\longrightarrow j$

0.25	0.25	1	4	1	1
4	4	0.25	0.25	0.25	6.25
6.25	6.25	1	0	1	**9**
1	1	0.25	2.25	0.25	2.25
1	1	0.25	2.25	0.25	2.25
1	1	0.25	2.25	0.25	2.25
0.25	0.25	4	9	4	0

D

$\uparrow i$

13.75					
13.5					
9.5	9.5	2.25	1.25	**2.25**	
3.25	3.25	1.25	3.25	3.25	5.5
2.25	2.25	1	3	3.25	5.5
1.25	1.25	0.75	3	3.25	5.5
0.25	0.5	4.5	13.5	17.5	17.5

A

Dynamic Time Warping

2. Compute accumulated cost matrix

- Name the accumulated cost matrix as `acuCost`. It has the same shape as `dists`.

Codes to determine the accumulated cost matrix

```
> acuCost      = np.zeros(dists.shape)
> acuCost[0,0]= dists[0,0]
```

$$A_{0,0} = D_{0,0}$$

starts from 1 because acuCost[0,0] has been filled with dists[0,0]

$$A_{0,j} = D_{0,j} + A_{0,j-1}$$

first row

```
> for j in range(1,dists.shape[1]):
      acuCost[0,j]     = dists[0,j]+acuCost[0,j-1]
```

$$A_{i,0} = D_{i,0} + A_{i-1,0}$$

first column

```
> for i in range(1,dists.shape[0]):
      acuCost[i,0]     = dists[i,0]+acuCost[i-1,0]
```

$$A_{i,j} = D_{i,j} + min(A_{i-1,j}, A_{i,j-1}, A_{i-1,j-1})$$

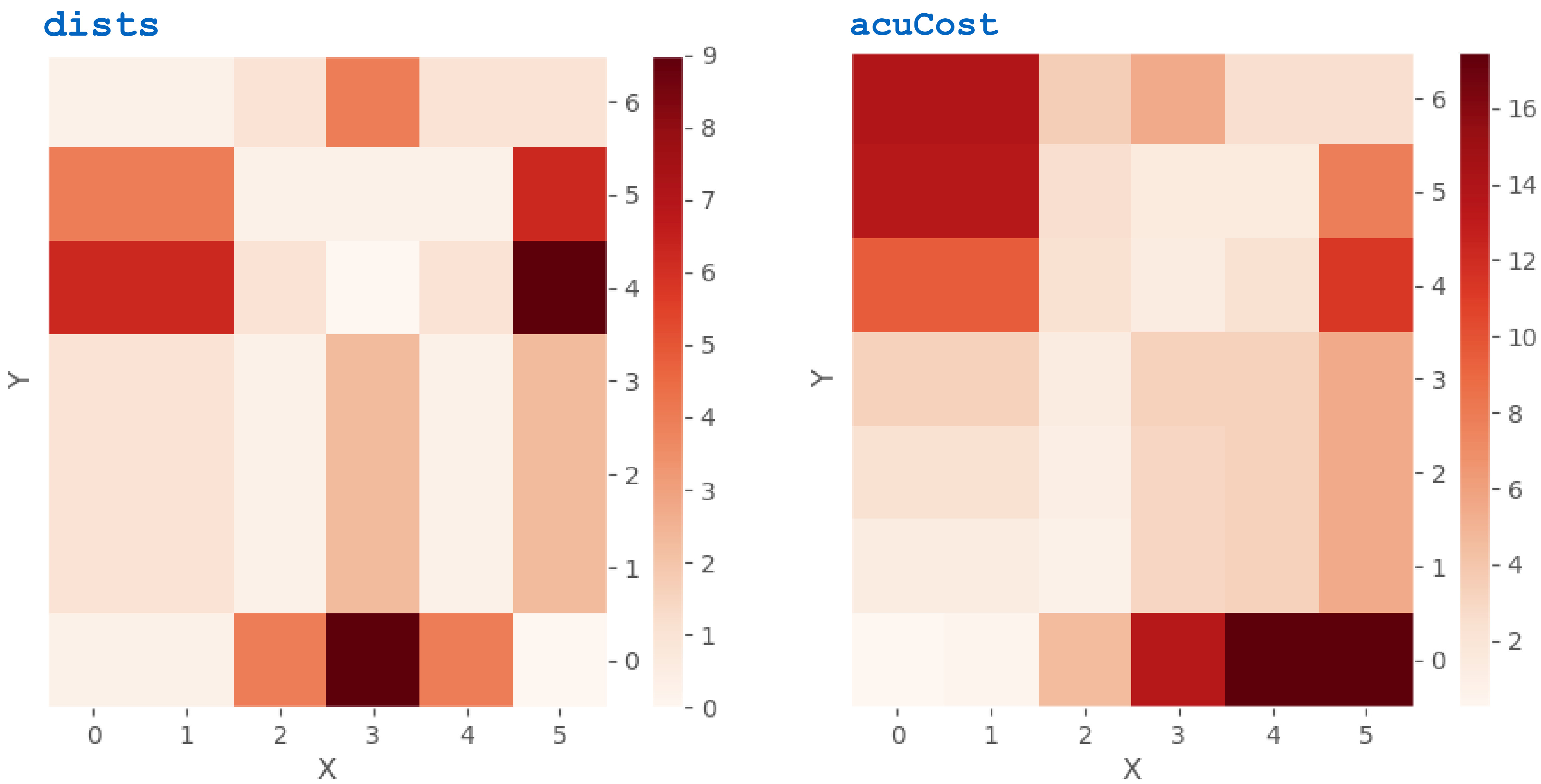
```
> for i in range(1,dists.shape[0]):
      for j in range(1,dists.shape[1]):
          acuCost[i,j]     = min(acuCost[i-1,j-1],
                                 acuCost[i-1,j],
                                 acuCost[i,j-1])+dists[i,j]
```

```
> pltDistances(acuCost,clrmap='Reds')
```

Dynamic Time Warping

2. Compute accumulated cost matrix

- Can you see the warping / optimal path?



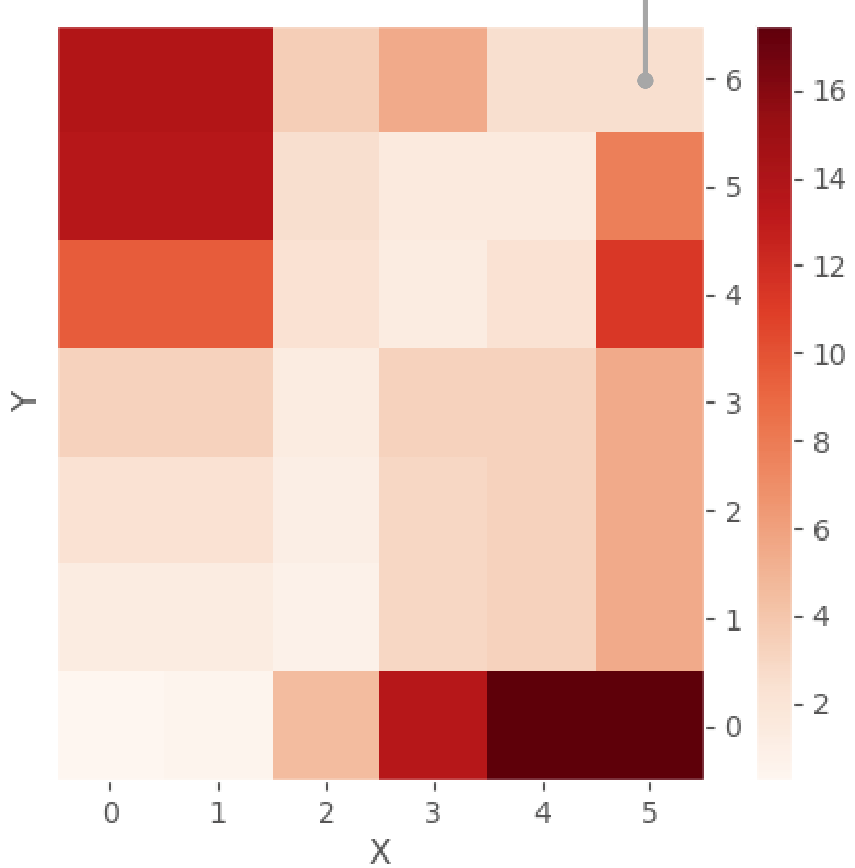
issm/m1.3/v1.0

Dynamic Time Warping

3. Search the optimal path

- Name the warping path as `path`.

it starts from here



a list; j stands for x axis, i stands for y axis

Wrapping process: Compare 3 neighboring squares and find the minimum cost square

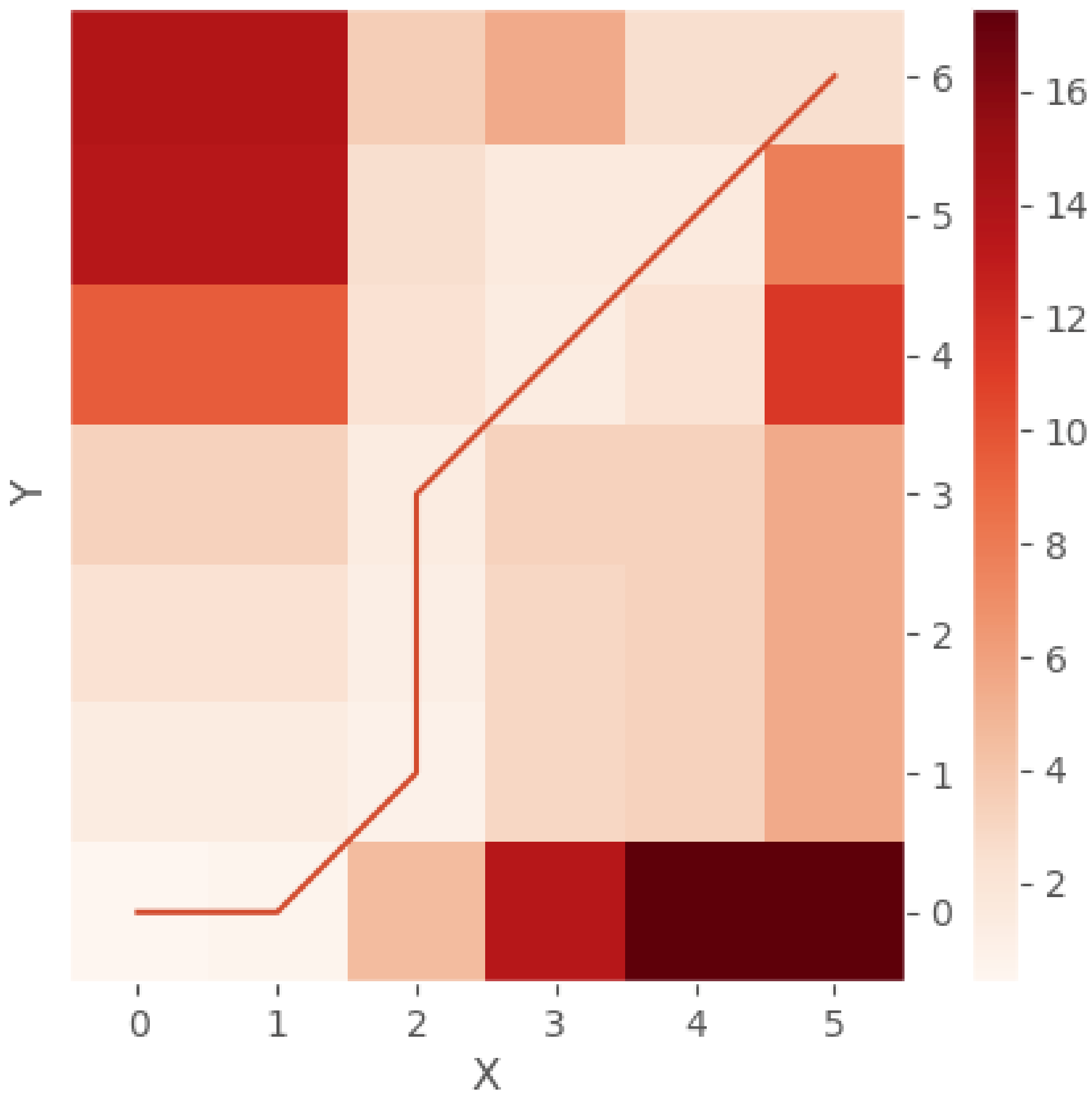
```
> i        = len(y)-1
> j        = len(x)-1
> path     = [[j,i]]
> while (i > 0) and (j > 0):
      if i==0:
          j   = j-1
      elif j==0:
          i   = i-1
      else:
          if acuCost[i-1,j] == min(acuCost[i-1,j-1],
                                   acuCost[i-1,j],
                                   acuCost[i,j-1]):
              i   = i-1
          elif acuCost[i,j-1] == min(acuCost[i-1,j-1],
                                     acuCost[i-1,j],
                                     acuCost[i,j-1]):
              j   = j-1
          else:
              i   = i-1
              j   = j-1
      path.append([j,i])
> path.append([0,0])
```

Dynamic Time Warping

3. Search the optimal path

- Create a function that plots the path

```
> def pltCostAndPath(acuCost,path,xlab="X",ylab="Y",clrmap="viridis"):
    px       = [pt[0] for pt in path]
    py       = [pt[1] for pt in path]

    imgplt   = pltDistances(acuCost,
                            xlab=xlab,
                            ylab=ylab,
                            clrmap=clrmap)
    plt.plot(px,py)

    return imgplt

> pltCostAndPath(acuCost,path,clrmap='Reds')
```



Now we want to plot the path; this code function helps us draw the path

Dynamic Time Warping

3. Search the optimal path

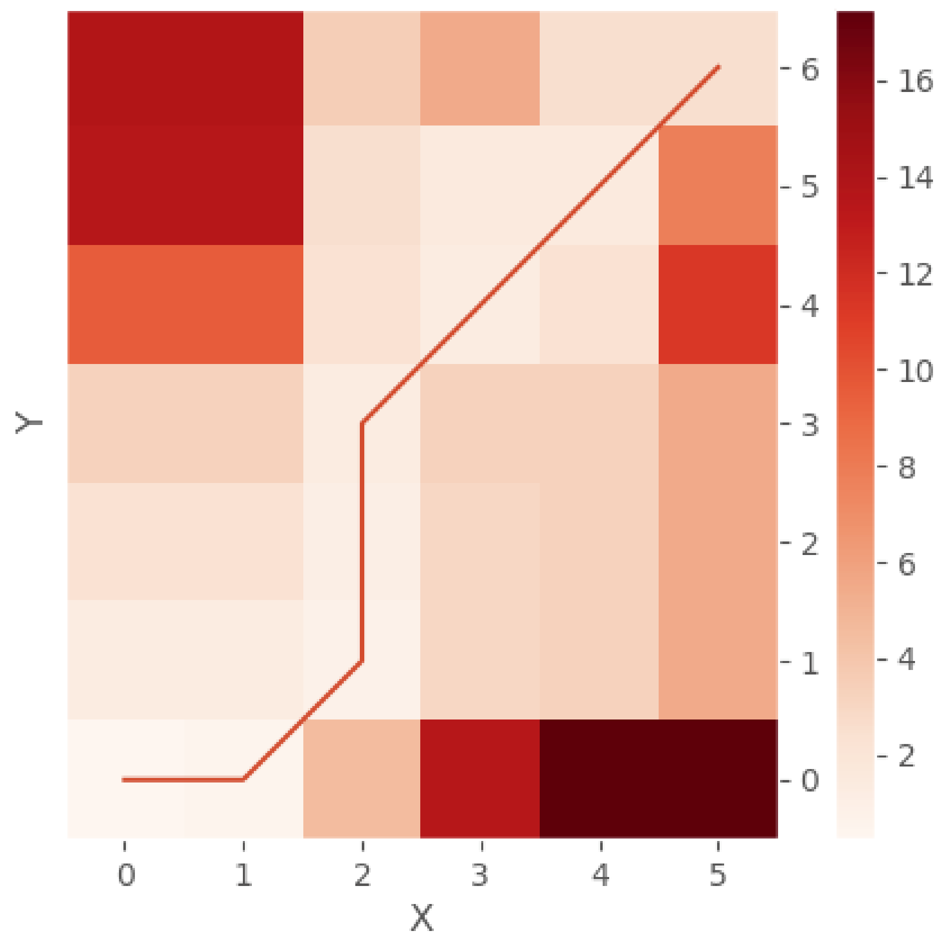
Rmb your accumulated cost is dependent on the distance:

A[i,j] = D[i,j] + ……

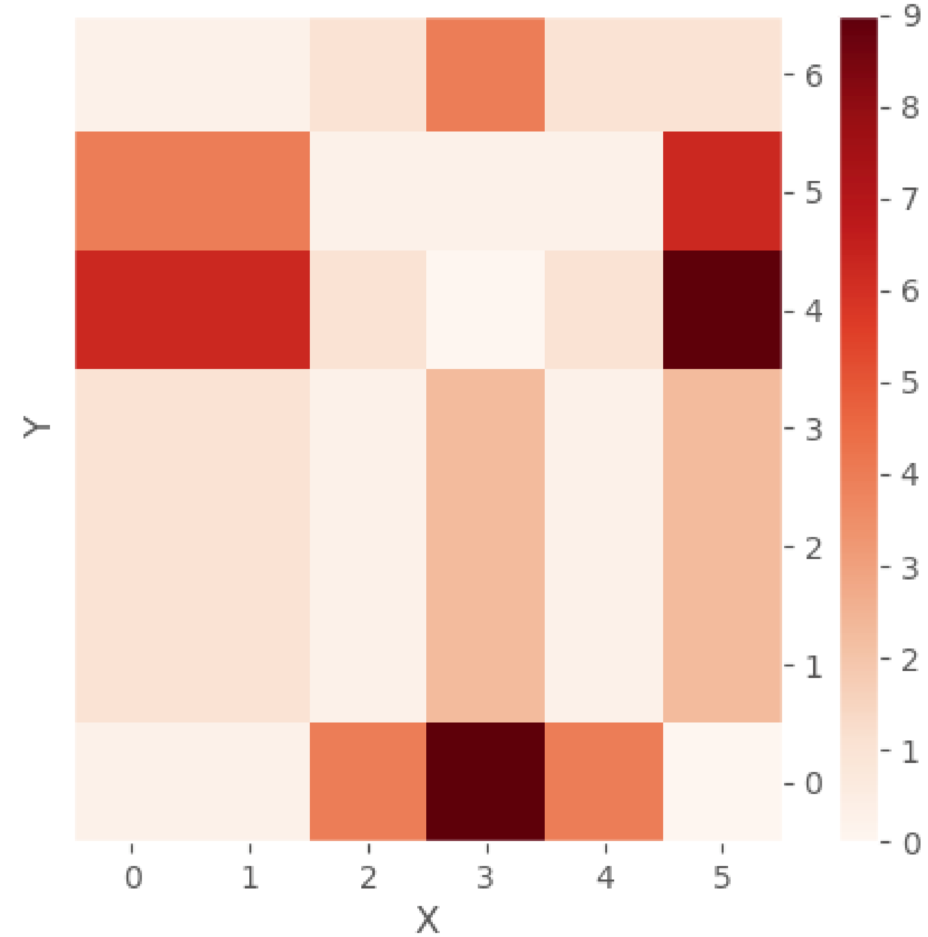
- Calculate the cost based on **`dists`**, which can be considered as a measure for similarity / distance

```
> cost          = 0
> for [j,i] in path:
        cost    = cost+dists[i,j]
> cost
: 2.5
```

acuCost



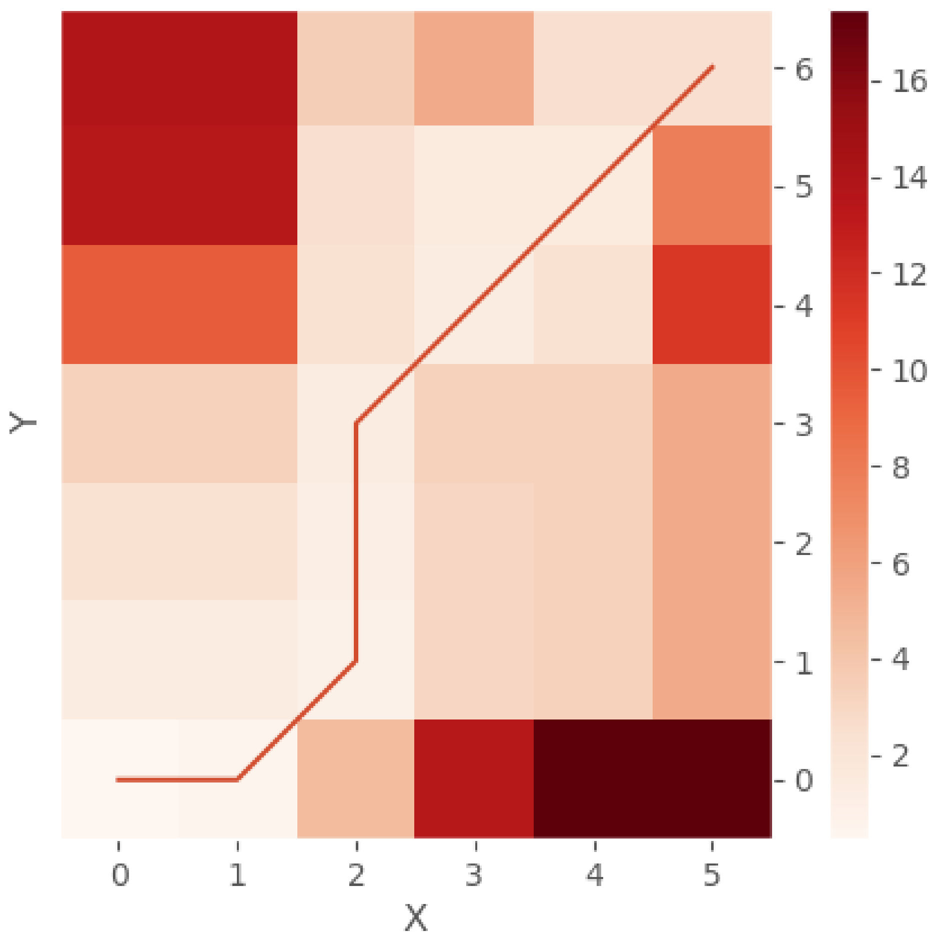
dists

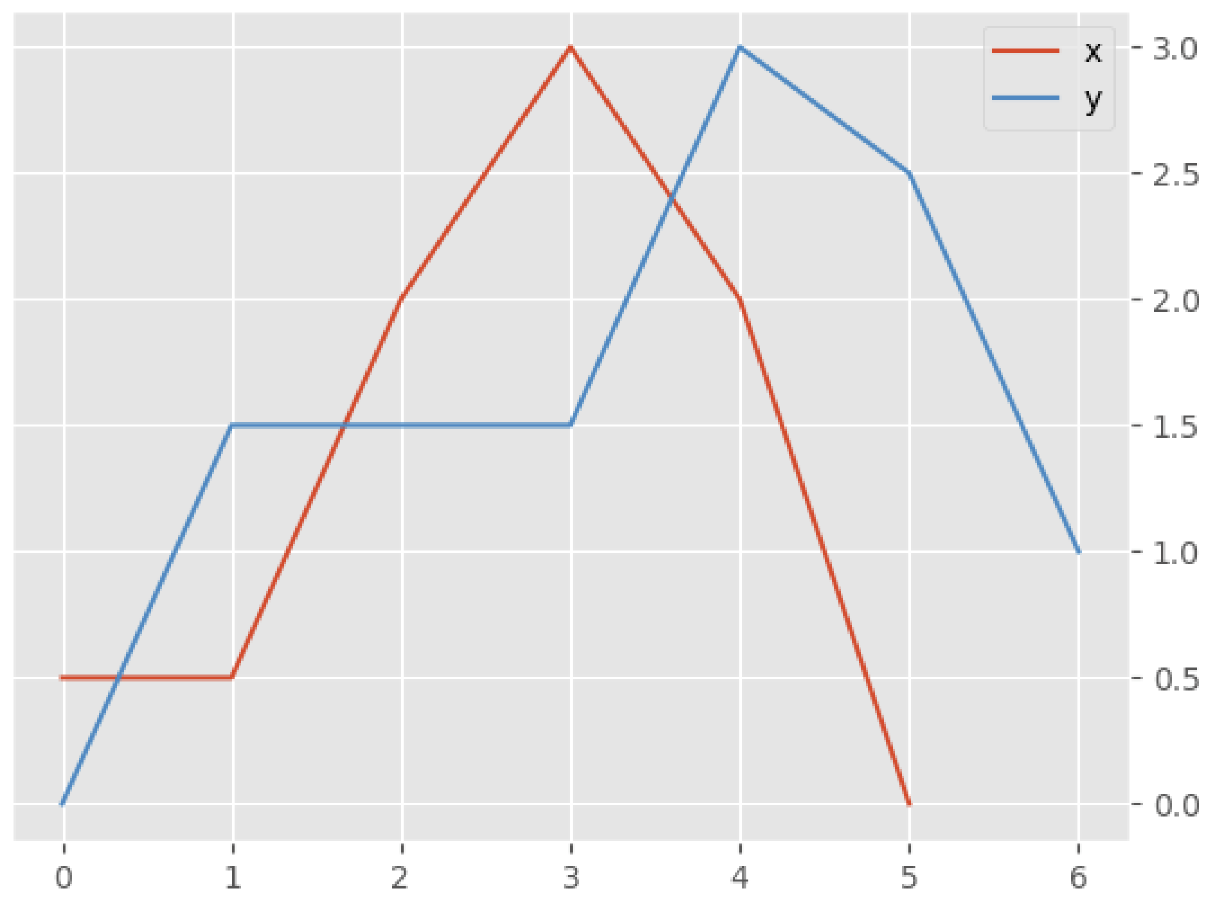


- The implication

How to illustrate the mapping between the 2 signals (Right) based on the wrapping path in the Cost Matrix (Left)?

`acuCost`



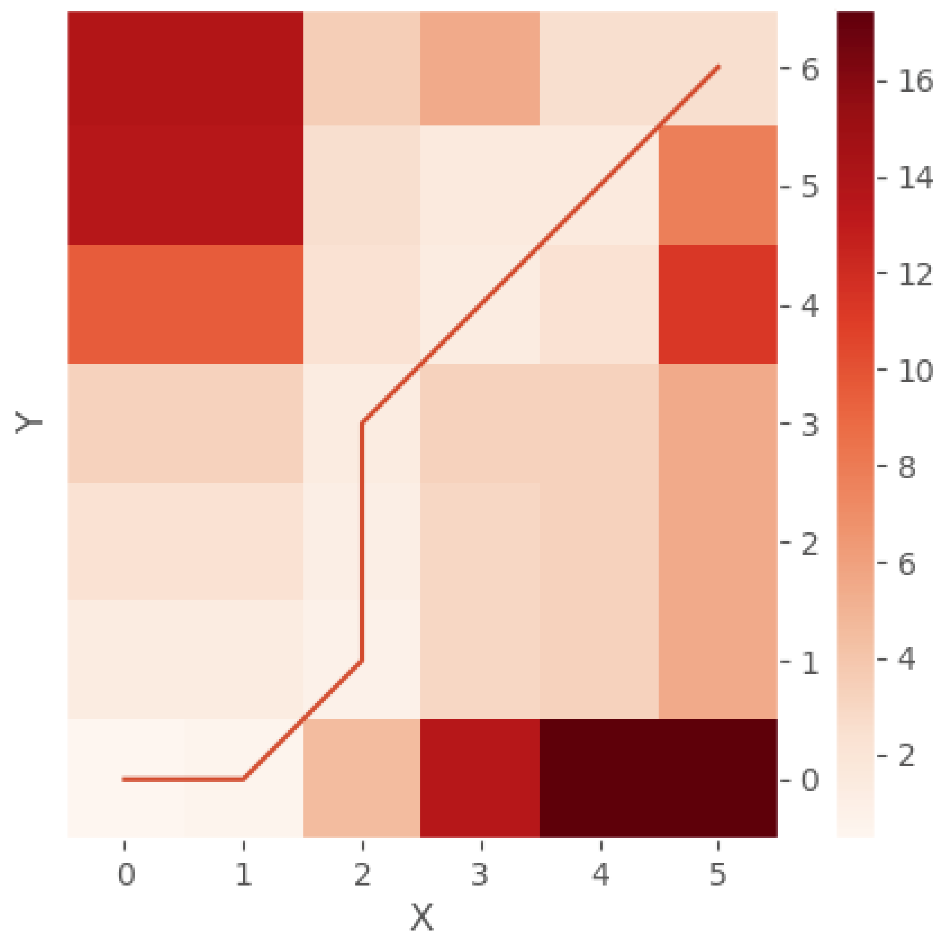


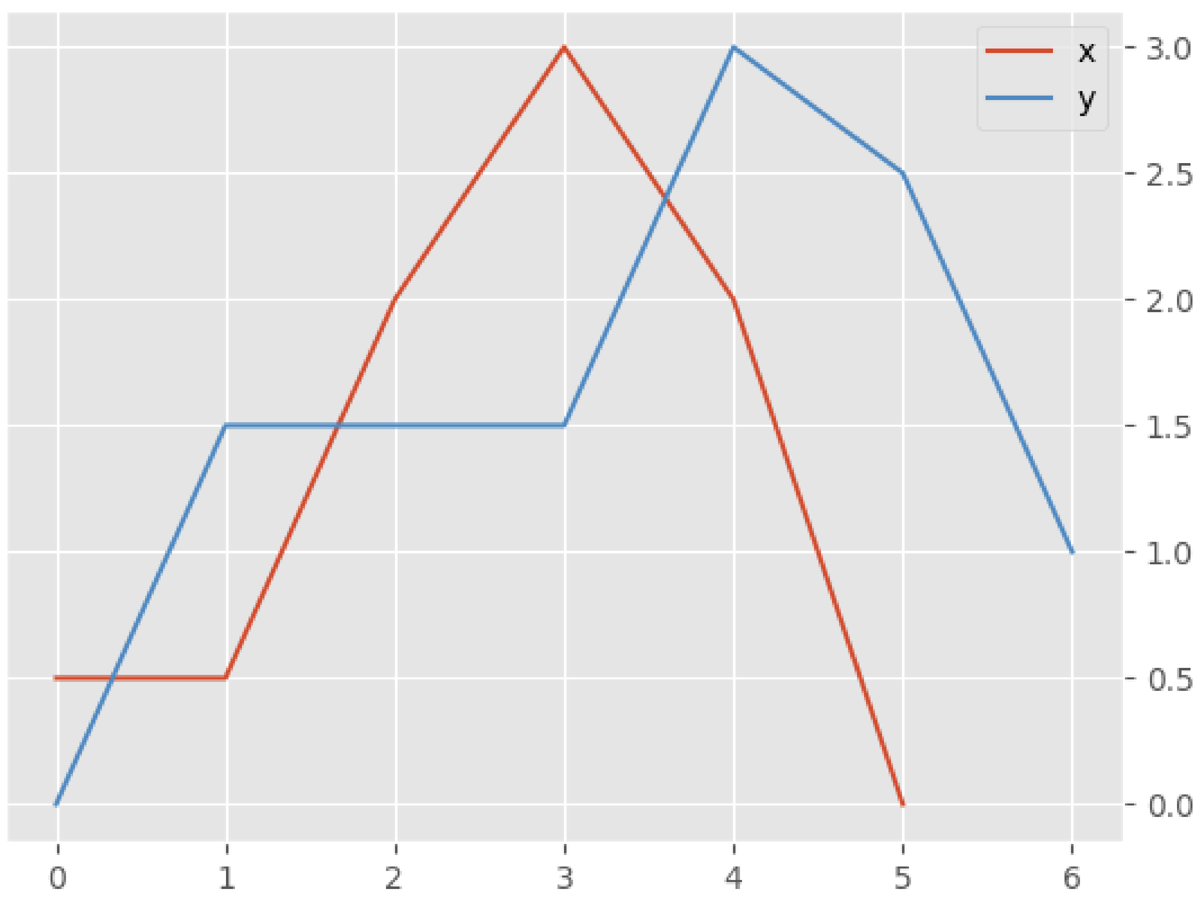
- The implication

How to plot a line that shows the correspondence between point 5 in X and point 6 in Y?

Ans: plt.plot([5, 6], [X[5], Y[6]])

`acuCost`





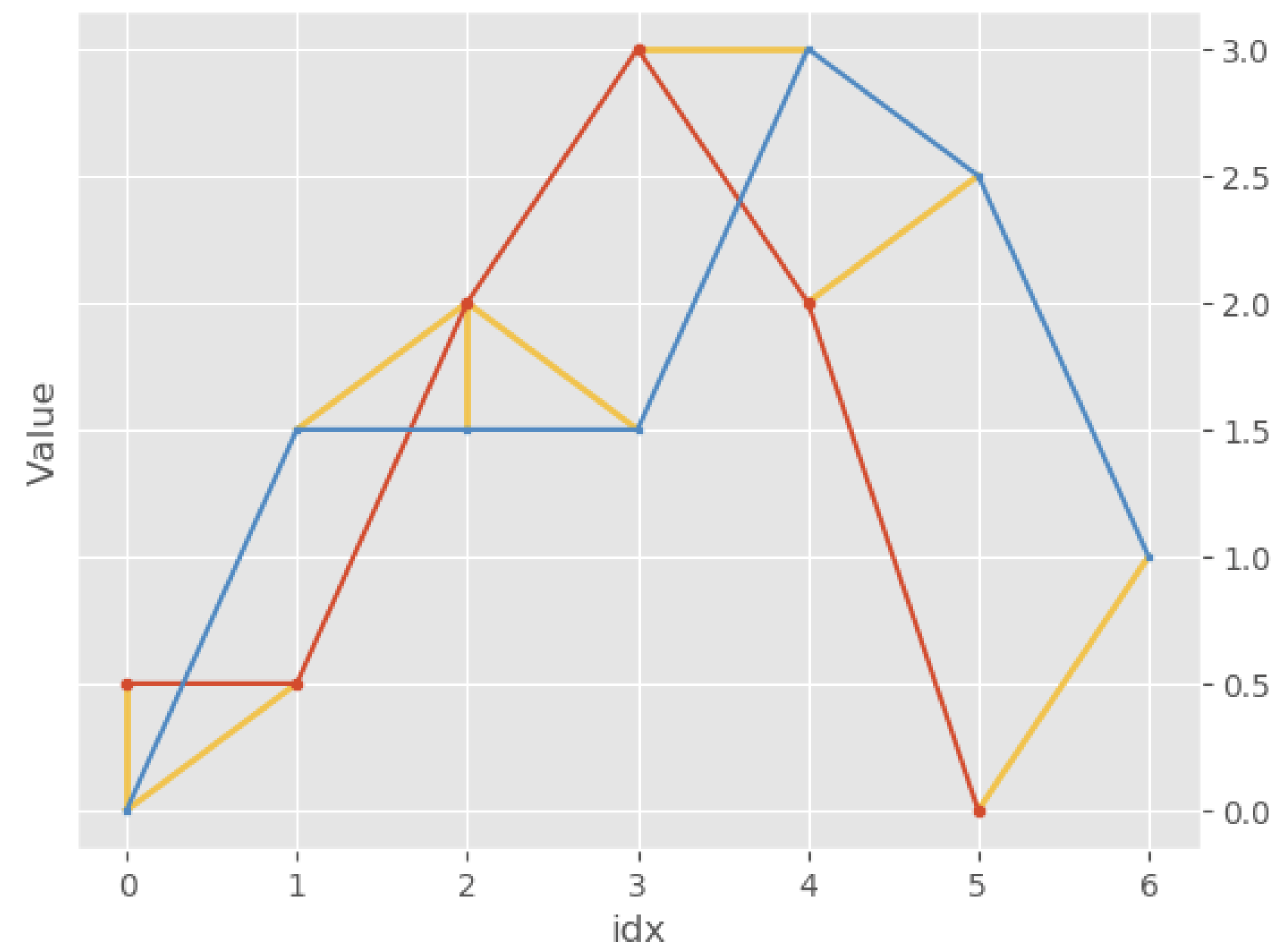
Dynamic Time Warping
3. Search the optimal path

- Plot the mapping of points between two signals

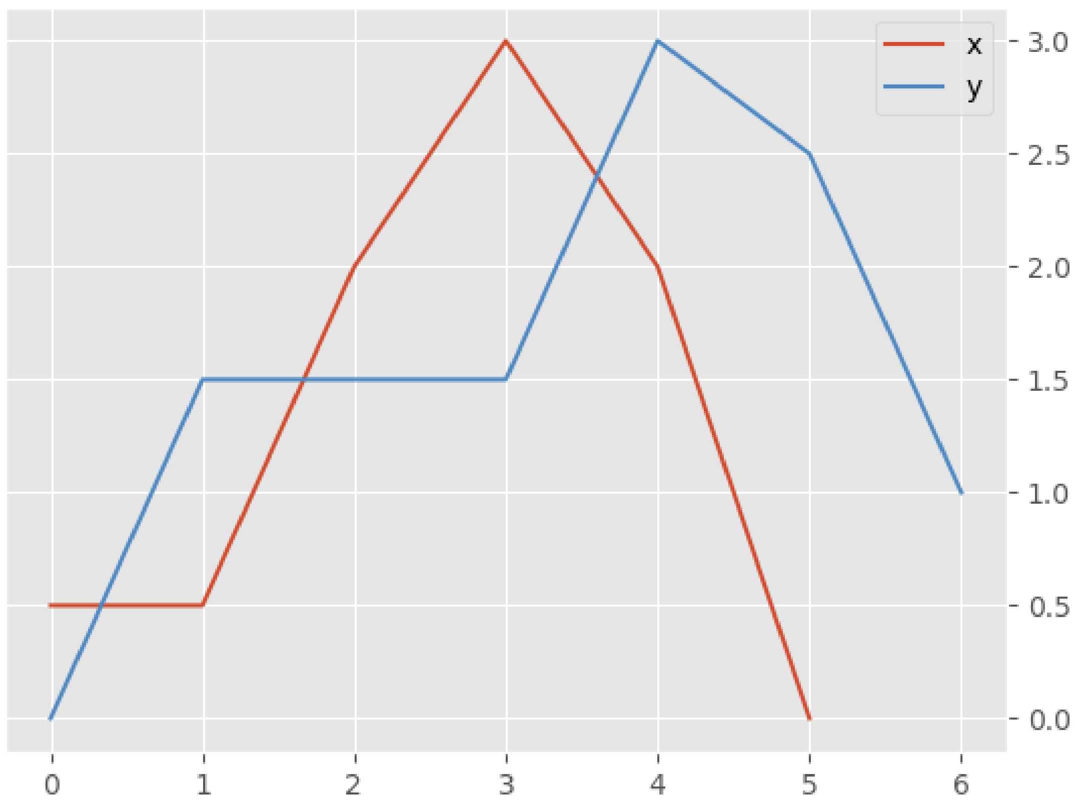
```
> def pltWarp(s1,s2,path,xlab="idx",ylab="Value"):
    imgplt      = plt.figure()

    for [idx1,idx2] in path:
        plt.plot([idx1,idx2],[s1[idx1],s2[idx2]],
                 color="C4",
                 linewidth=2)
    plt.plot(s1,
             'o-',
             color="C0",
             markersize=3)
    plt.plot(s2,
             's-',
             color="C1",
             markersize=2)
    plt.xlabel(xlab)
    plt.ylabel(ylab)

    return imgplt

> pltWarp(x,y,path)
```

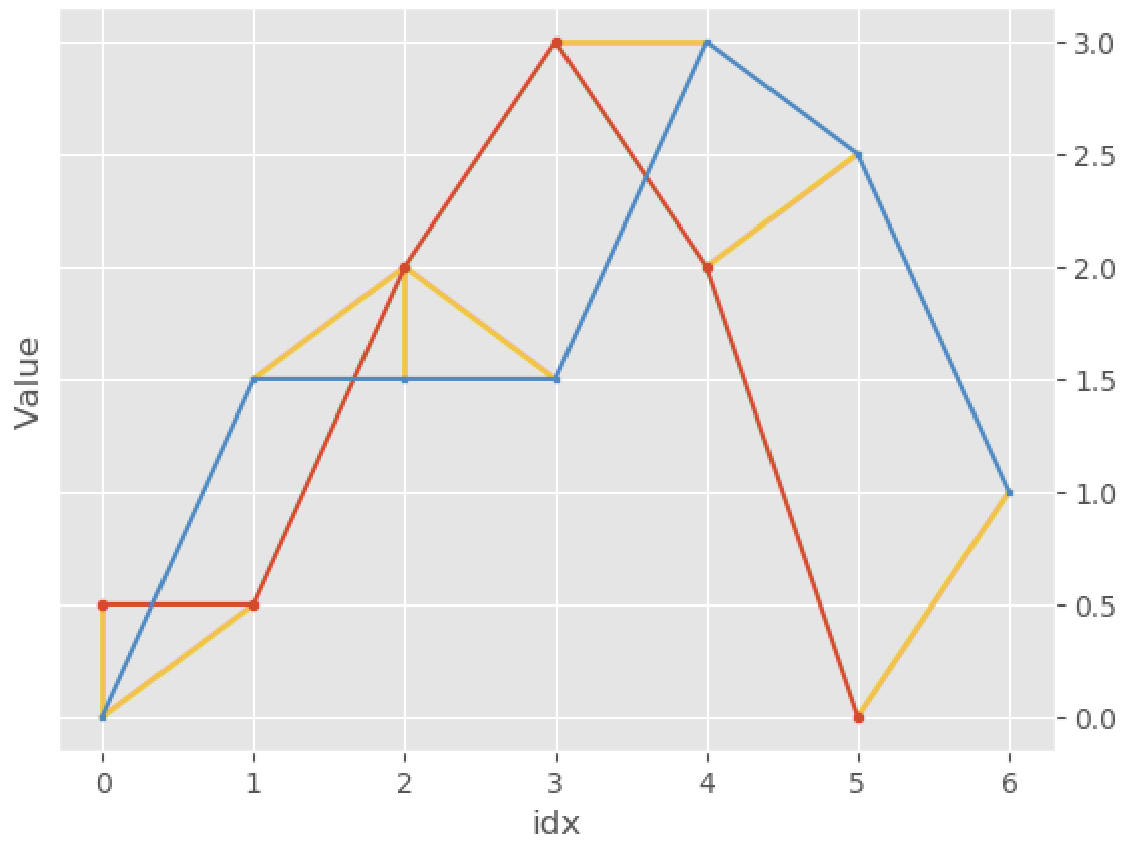
idx1 and idx2 are the interested time parameters

Plot the connections between s1 and s2 (yellow lines)



Dynamic Time Warping

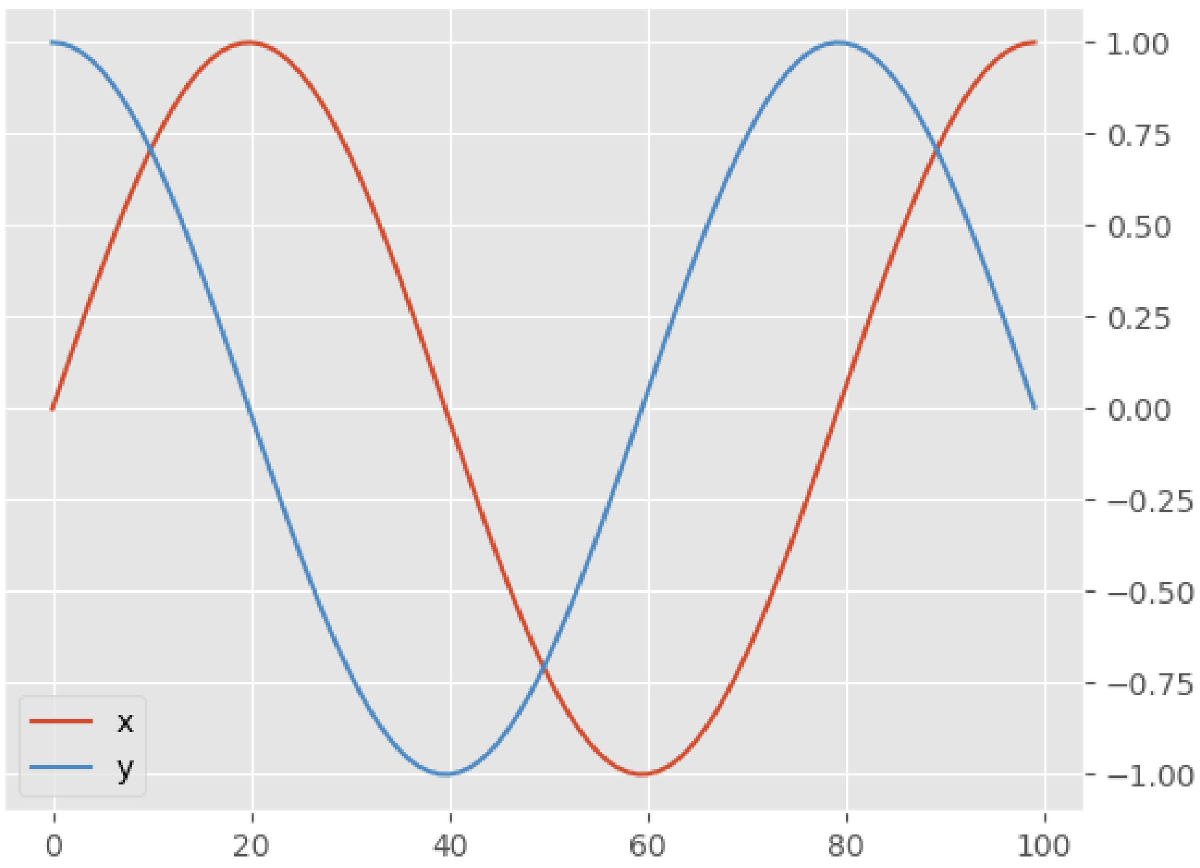




Dynamic Time Warping

Another example

Another example with 2 different signals



- Define two signals as:

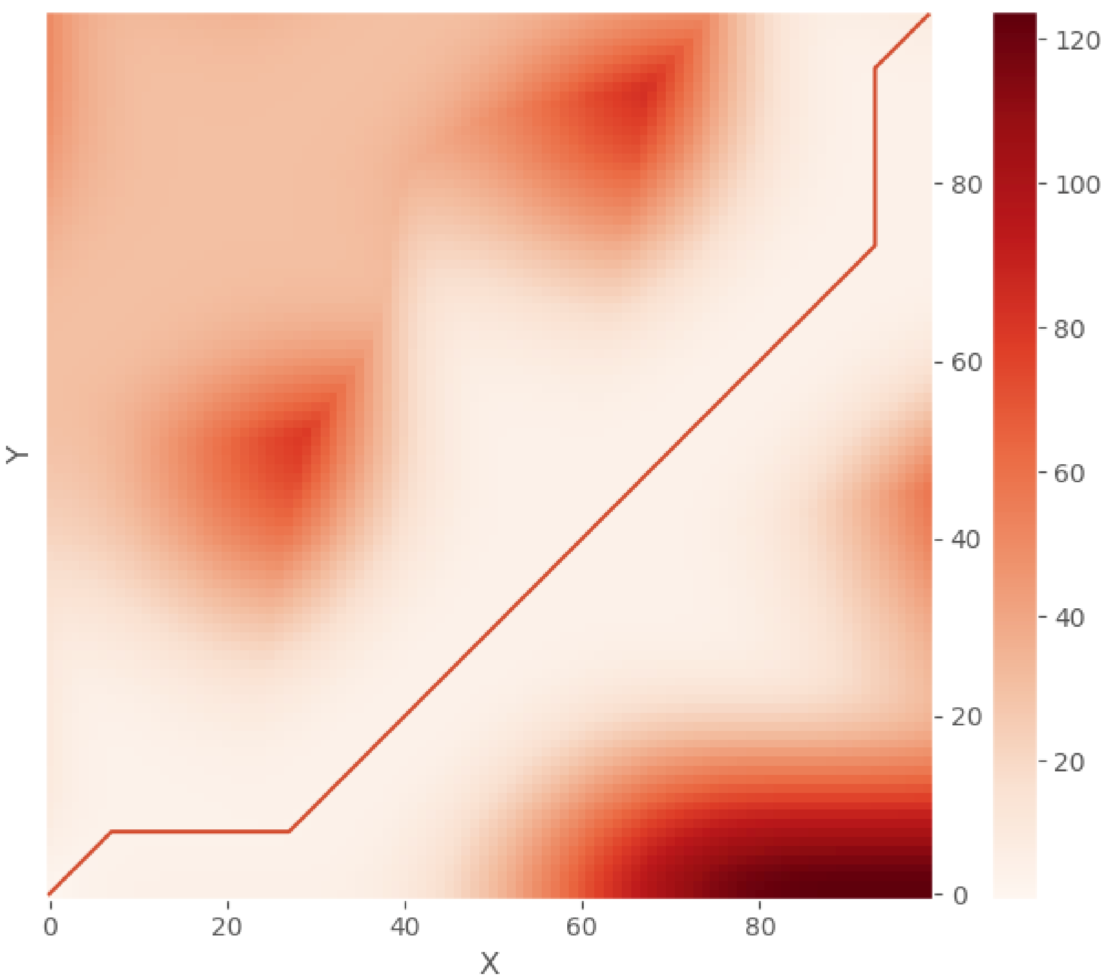
```
> x = np.sin(np.linspace(0,7.85,100))
> y = np.cos(np.linspace(0,7.85,100))
```

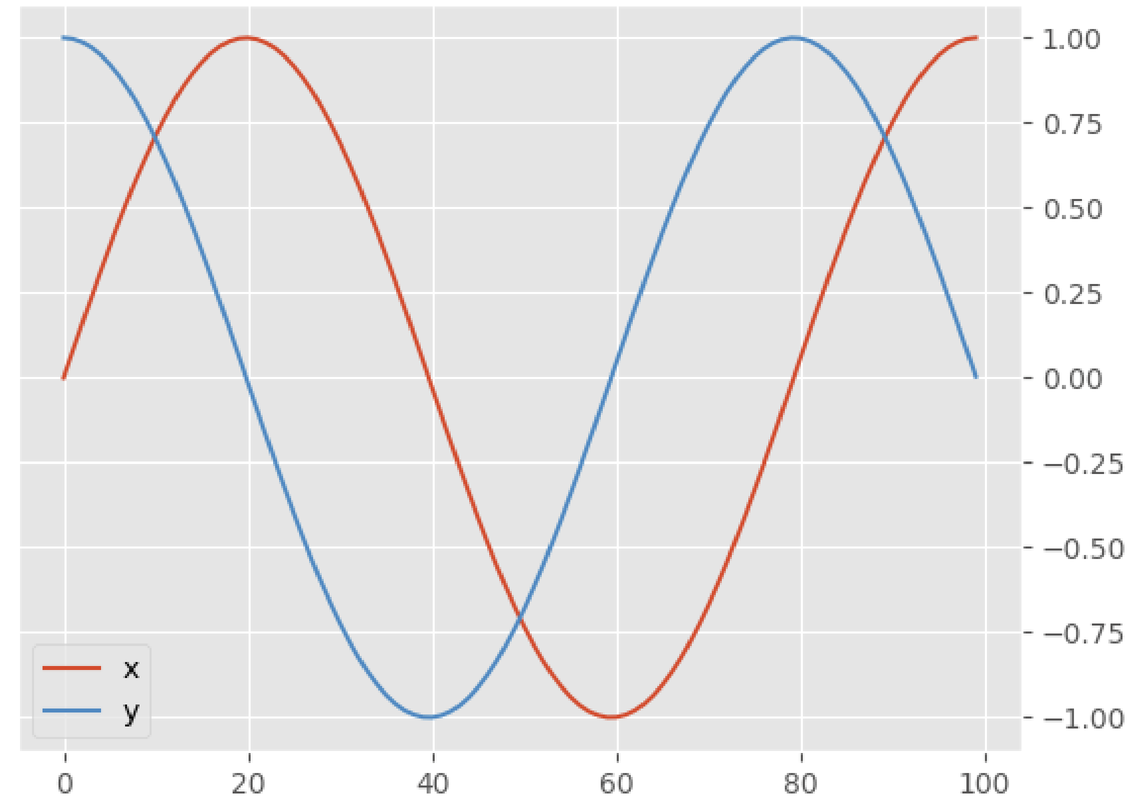
- Plot the two signals

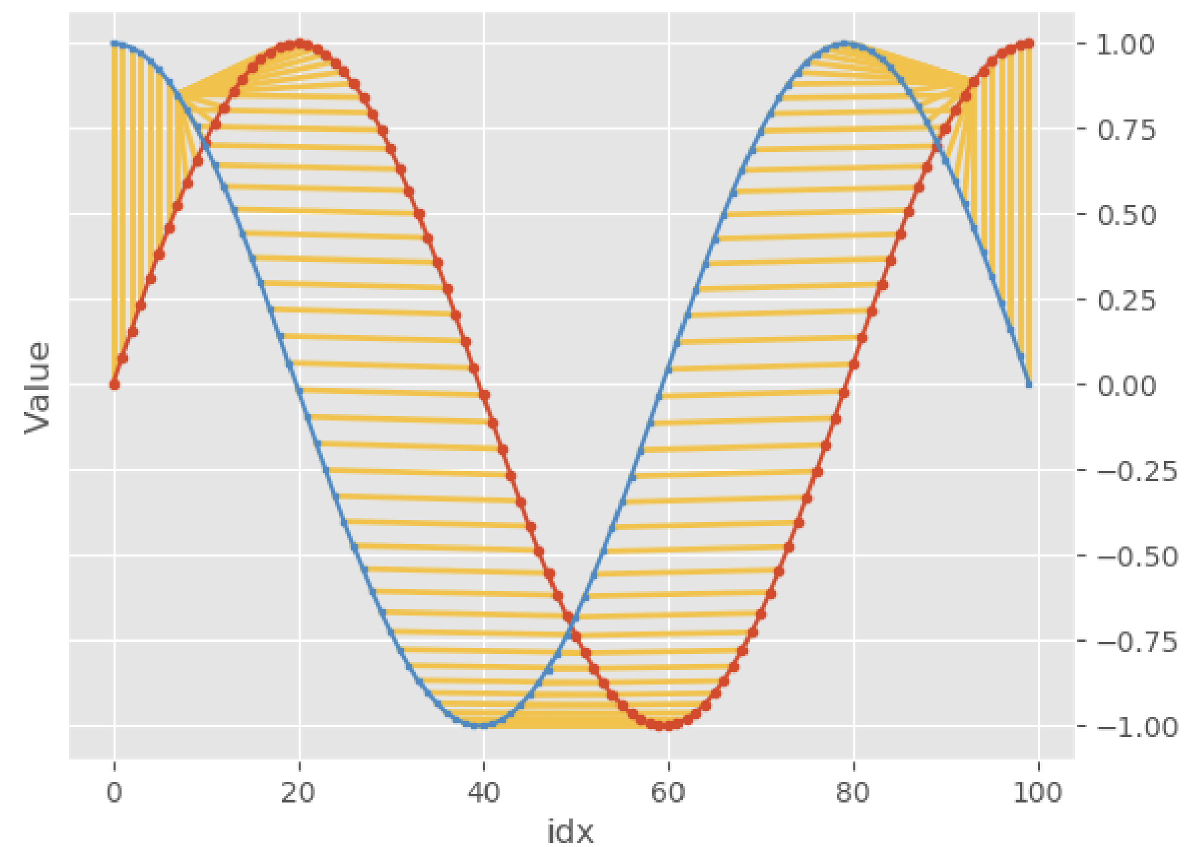
```
> plt.figure()
> plt.plot(x,
           color="C0",
           label='x')
> plt.plot(y,
           color="C1",
           label='y')
> plt.legend()
```

Dynamic Time Warping

Another example







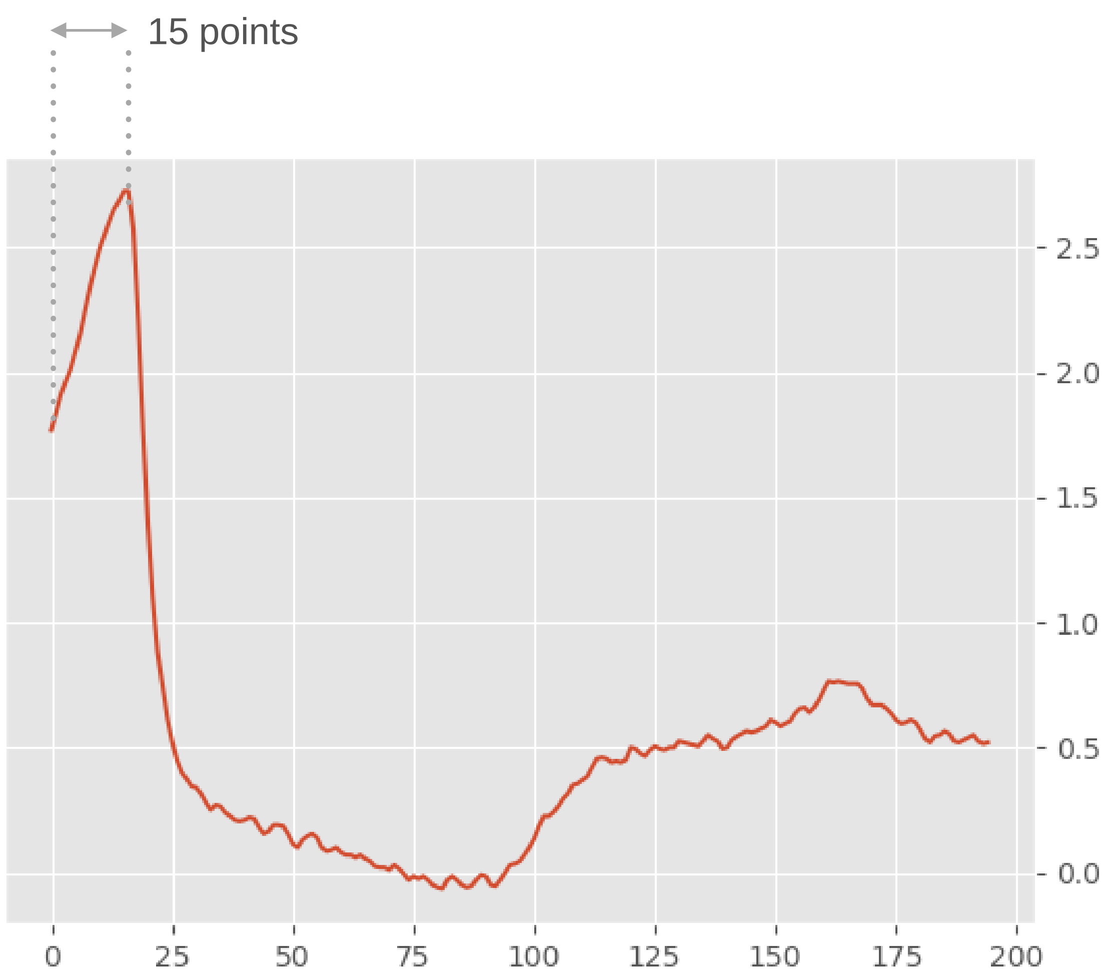
Back to the problem

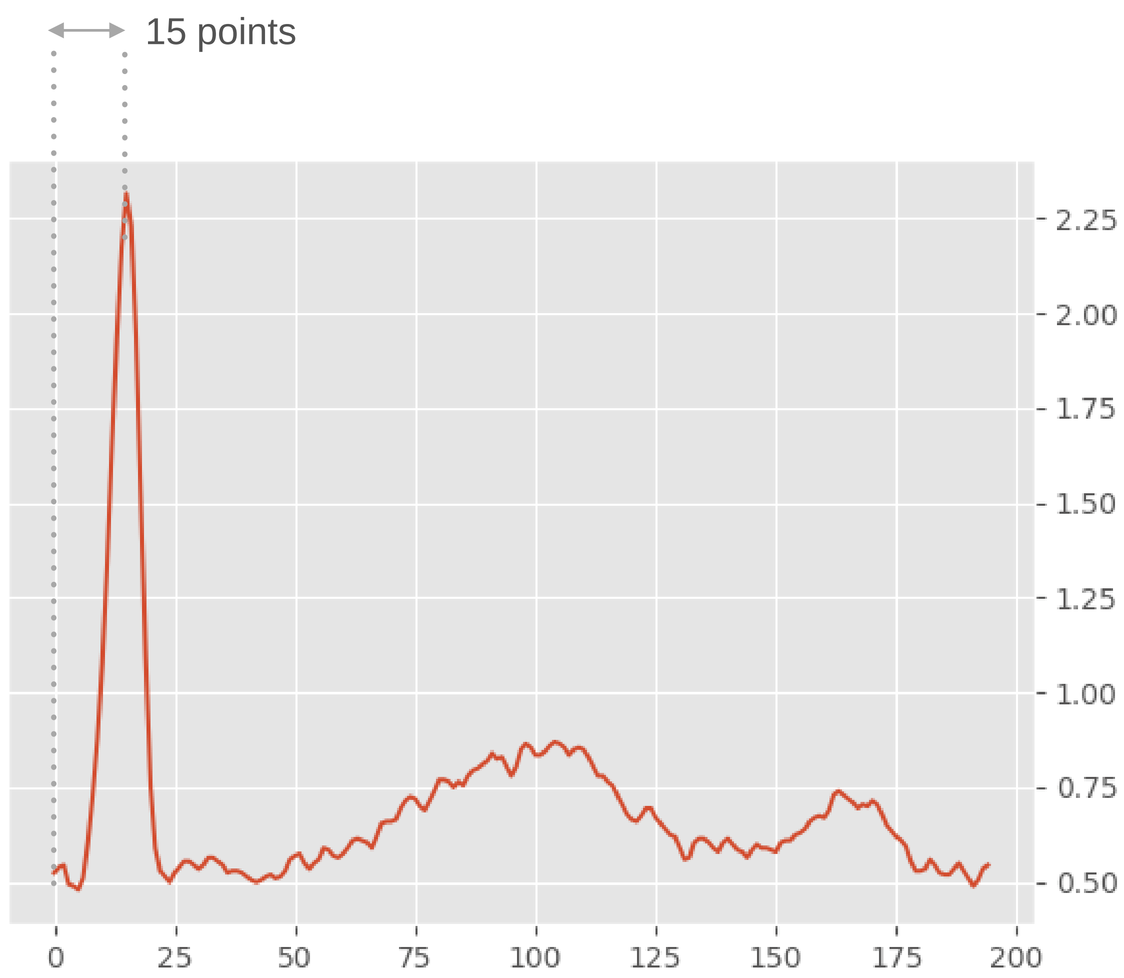
How to start?

Use peak (i.e. `findpeak`) to estimate start of each ECG heartbeat by shifting 15 points to left from peak

- Before we compute similarity, must segment individual heartbeat signal

- With signal segmented, perform DTW





Workshop
To start

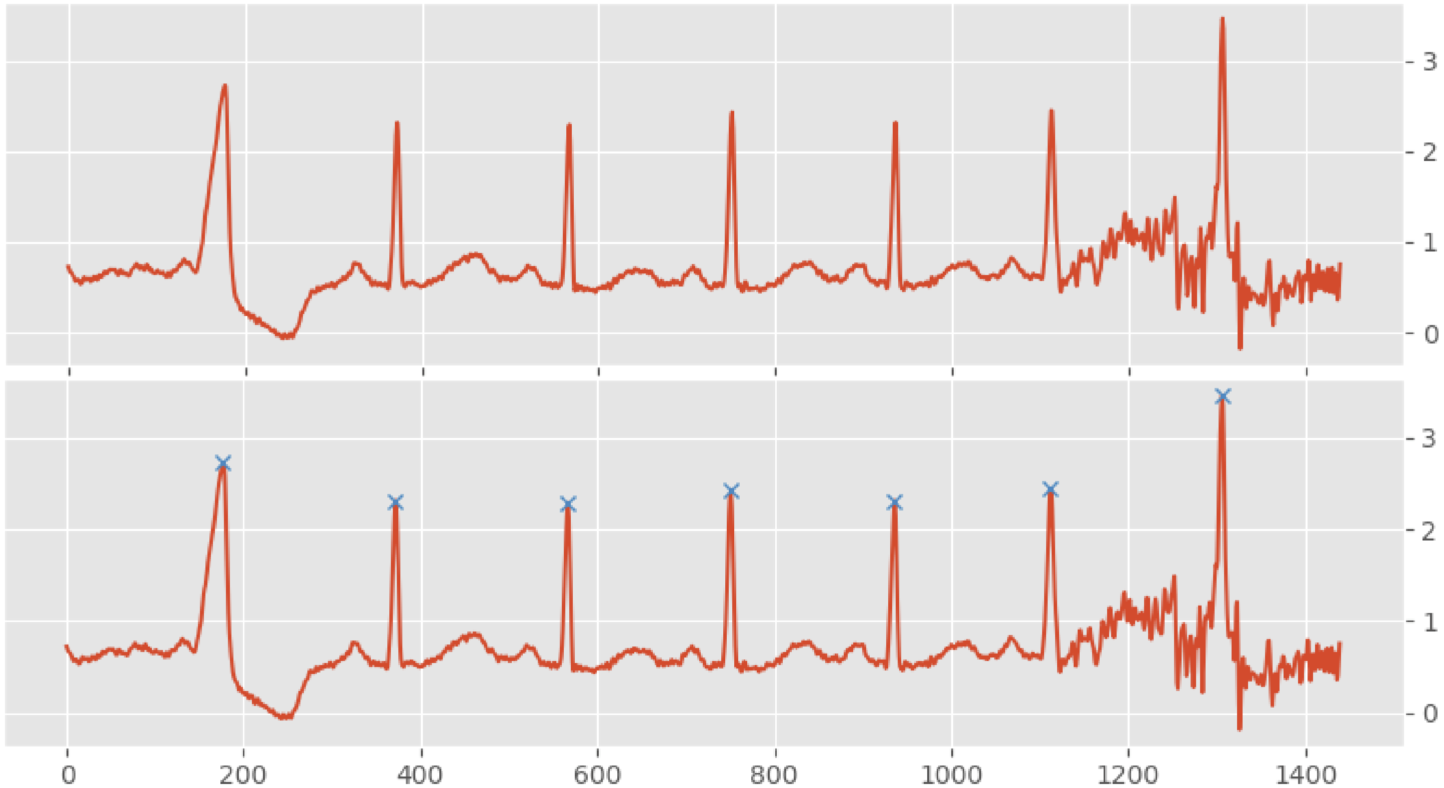
You need to complete a code function to chop (segment) each ECG heartbeat from the complete signal!

- Load the data

```
> l2D      = pd.read_csv('ecg2D.csv',
                        header=None)
> ECGs     = l2D[1].values
```

- Create a function with the below signature. The output is a list consists of all the ECG segments in a ECG signal

```
def extractECG(ecg,pks,offset=15):
```



General Procedures:

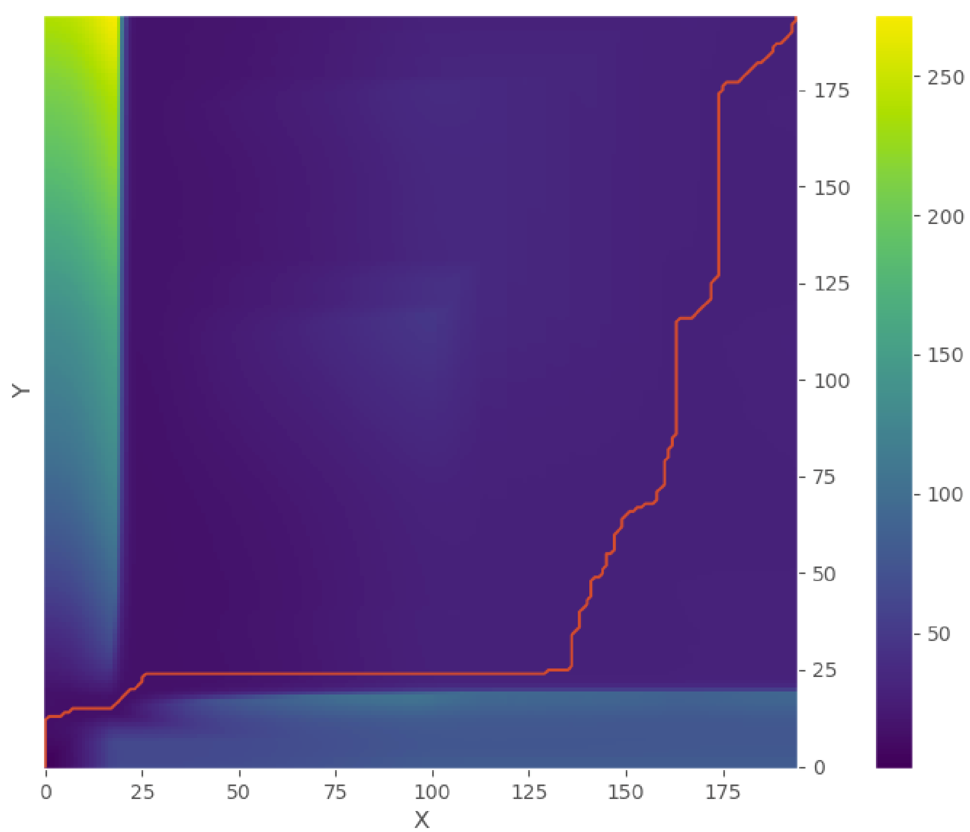
1. **Load data**
2. **Perform peak detection**
3. **Extract segments based on the detected peaks**

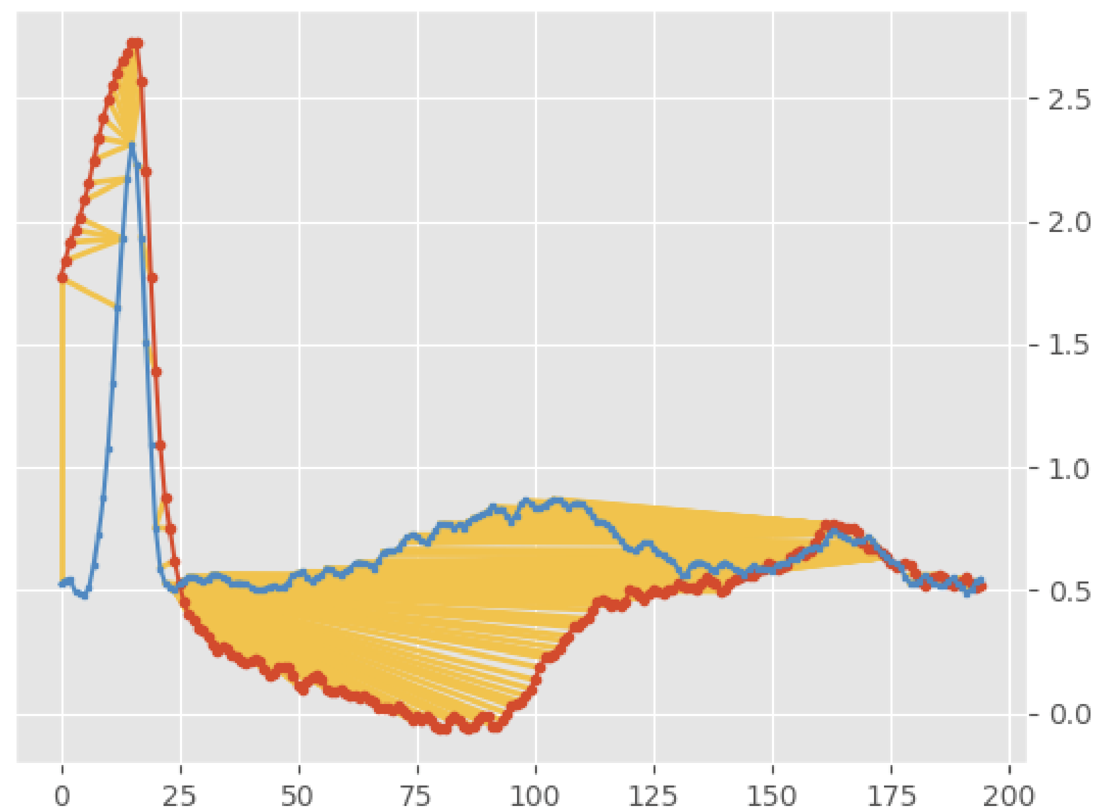
Workshop

Compute the accumulated costs, plot the optimal paths and warp

If you are done early, try on other signals (i.e. other than the 2nd column of the `ecg2D` data)

- Make the comparisons between segment
  - 1 and 2
  - 2 and 3
  - 2 and 6





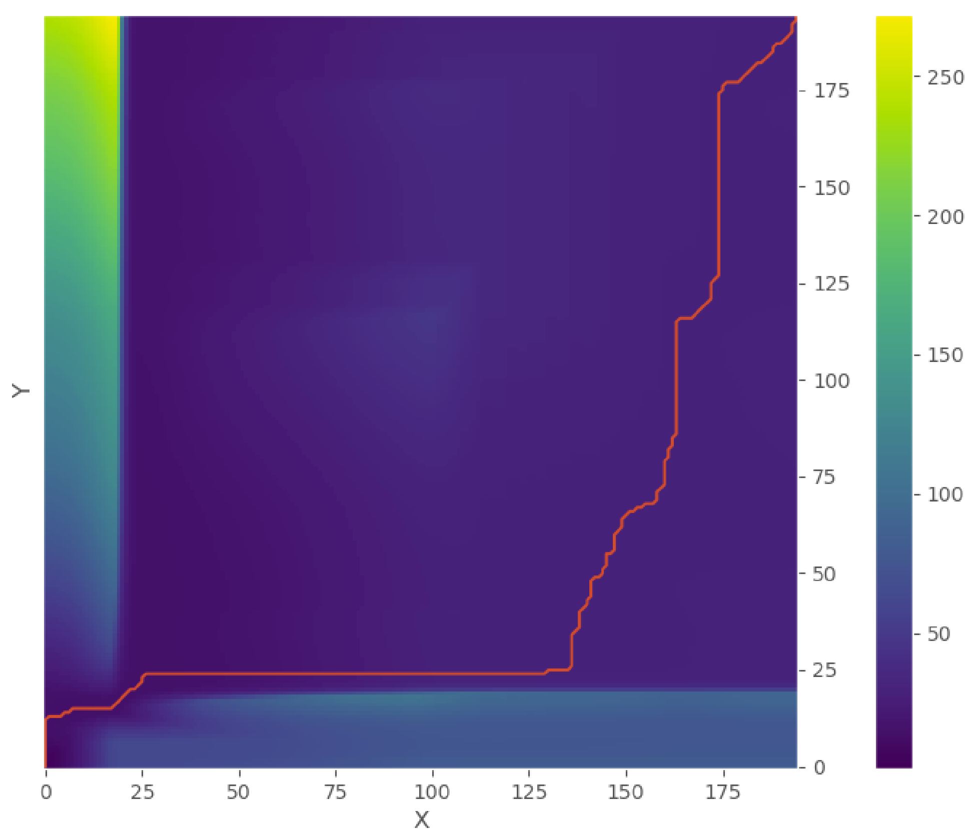
Workshop Hints for `extractECG` function

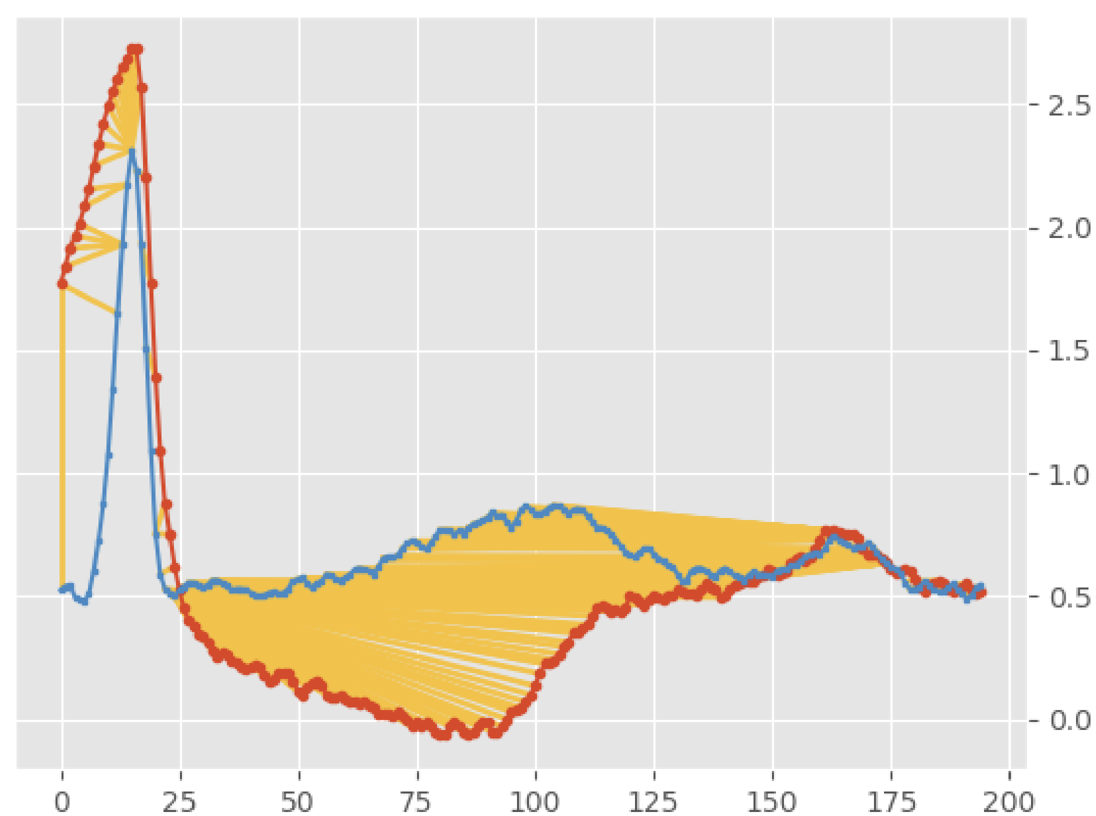
Tips how to chop the ecg:

Assume, you have all the peaks being identified, and ‘pks’ is the list to hold all the position of peaks. Assume ‘ecg’ is the ECG signals. Then you need to go through a “for-loop”, for each iteration, you do:

```
Seg = ecg[(pks[i]-offset):(pks[i+1]-offset)]
```

Another Tip: You need to ignore the last peak





Workshop Hints for `extractECG` function

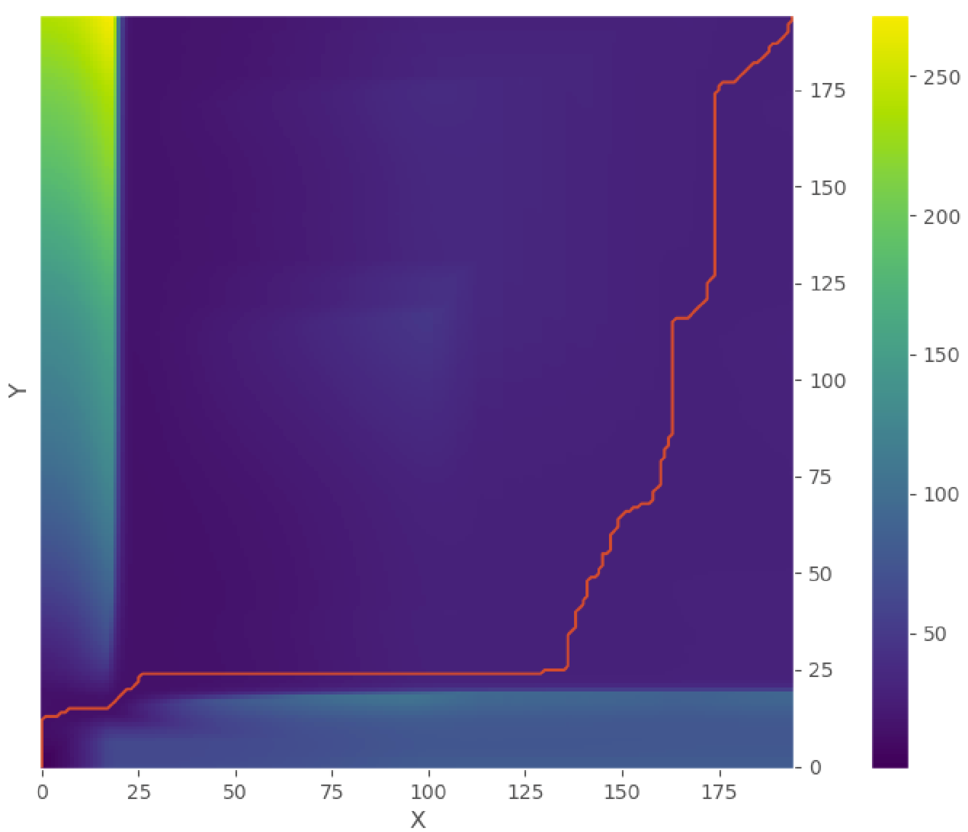
Tips how to chop the ecg:

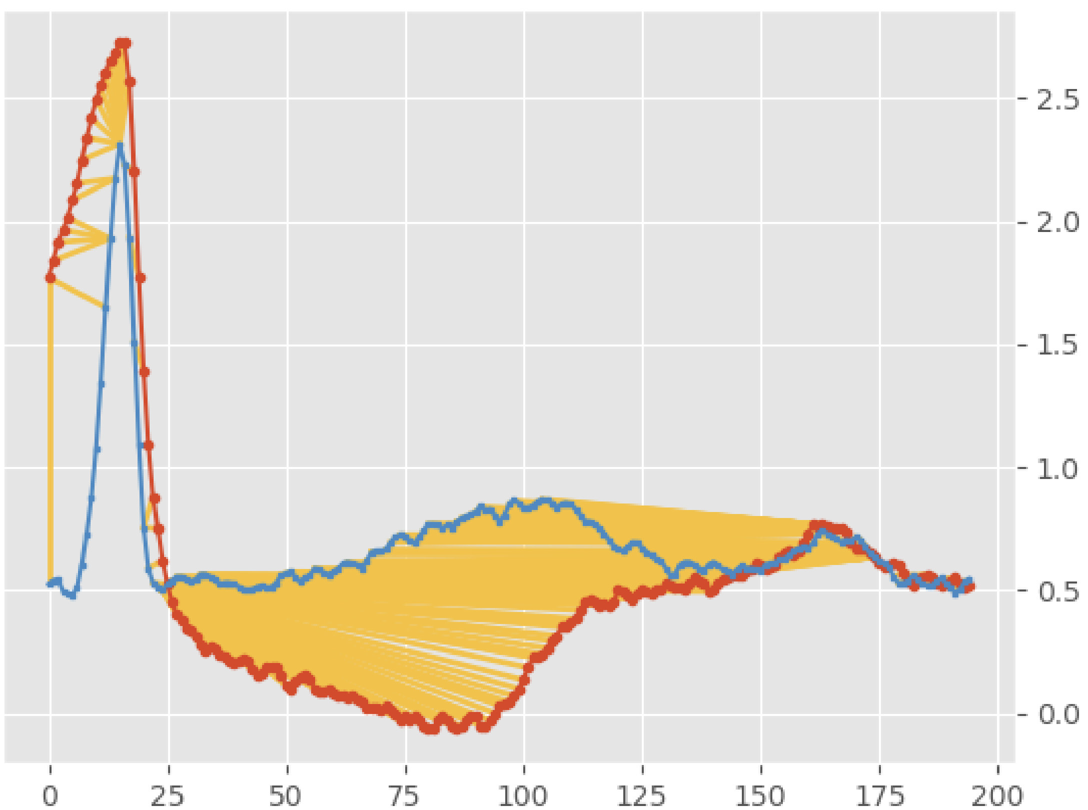
```
def extractECG(ecg,pks,offset=15):
    segs    = [ ]
    if pks[0]-offset < 0:
        start   = 1
    else:
        start   = 0

    for i in range(start,len(pks)-1):
        seg = ecg[(pks[i]-offset):(pks[i+1]-offset)]
        segs.append(seg)

    return segs

EcgSegs = extractECG(ECGs,Pks)
```

Workshop Hints for Performing DTW on ECG segments

Example for comparing between segments 1 and 2:

```
dist = computeDists(segs[0],segs[1])
acuCost = computeAcuCost(dist)
(path,cost) = doDTW(segs[0], segs[1], dist, acuCost)
pltCostAndPath(acuCost, path)
pltWarp (segs[0], segs[1], path)
```





Appendix: Comparison among Various Methods

To measure similarity between signals

	Pros	Cons
Euclidean/Manhattan distance	Easy to implement; straightforward	Only works if the signals to be compared are of same length and preferably similar shape
DTW	Not necessary for signals to be of same length and shape; Computes faster than LCSS	Heavy computational burden (worse than Euclidean/Manhattan distance method); Unable to work if one of the signals is a partial type
Longest common subsequence (LCSS)	Not necessary for signals to be of same length and shape; Allows for the use of partial and noisy signals	Heavy computational burden; Computes slower than DTW
Developed Longest Common Subsequence (DLCSS)	Not necessary for signals to be of same length and shape; Allows for the use of partial and noisy signals; Most accurate among the other methods	Heavy computational burden; Computes slower than DTW and LCSS