CS598PS - Machine Learning for Signal Processing

Detection and Template Matching

17 September 2020

A change in thinking

- So far we did feature extraction
  - “Unsupervised learning”
- Now we’ll do detection and classification
  - “Supervised learning”

Today's lecture

- Detection of elements of interest
- Template matching / matched filters
  - Obtaining invariance/robustness
  - Efficient matching
  - Usage cases
- Probabilistic interpretation

Detection

- Find the presence of a predefined signal
  - Face/pedestrian/car detection
  - Speech, heartbeat, gunshot detection
  - Earthquakes, quasars, aliens, …
- Many ways to go about it
  - We'll start from the basics

A simple example

- Can we detect a known face in a collection?

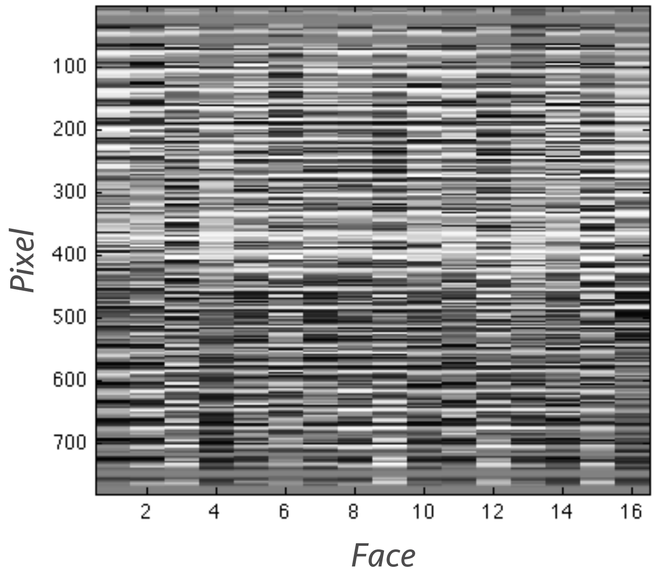
Query

Search set

Vectorize the data (and scale to unit norm)

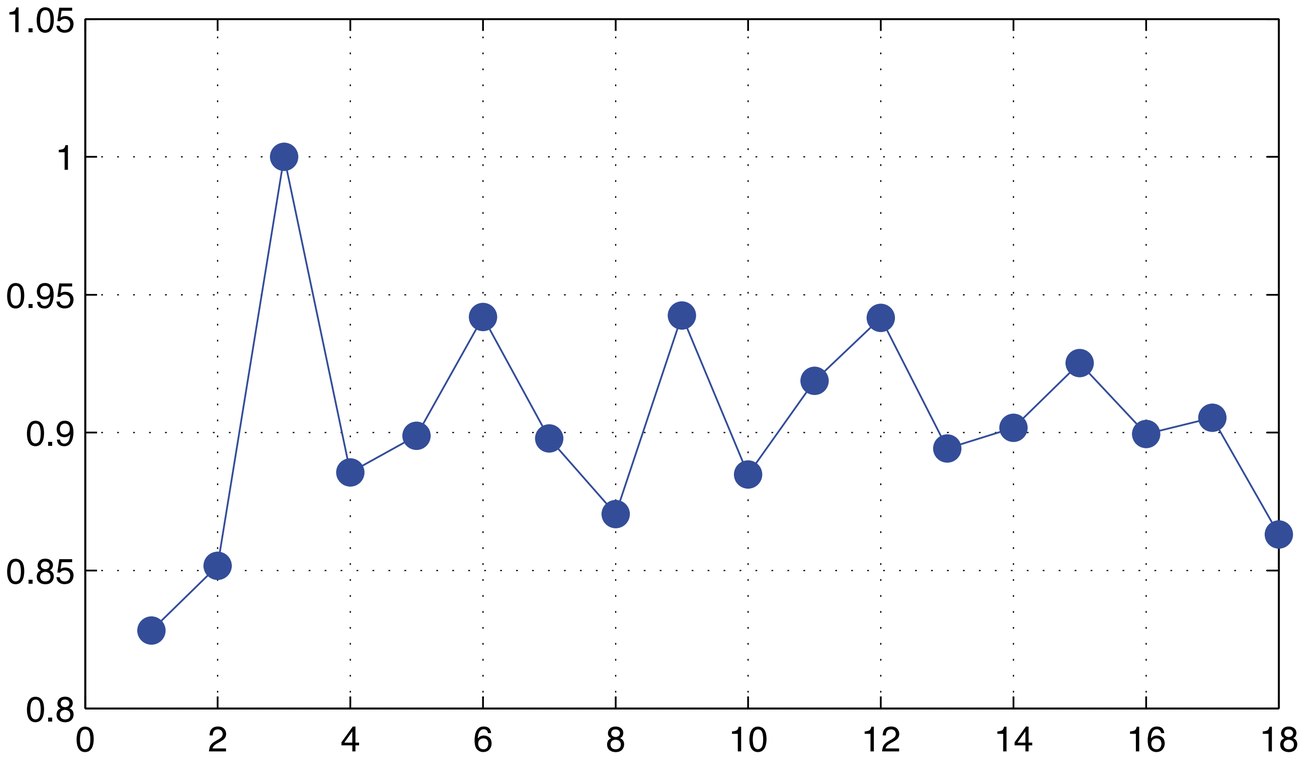
$$\mathbf{x}^\top = \text{vec}(\quad)^\top =$$

$$\mathbf{Y} = \text{vec}(\quad) =$$



And get their dot product

$$\mathbf{x}^\top \cdot \mathbf{Y} =$$



Dot product for matching

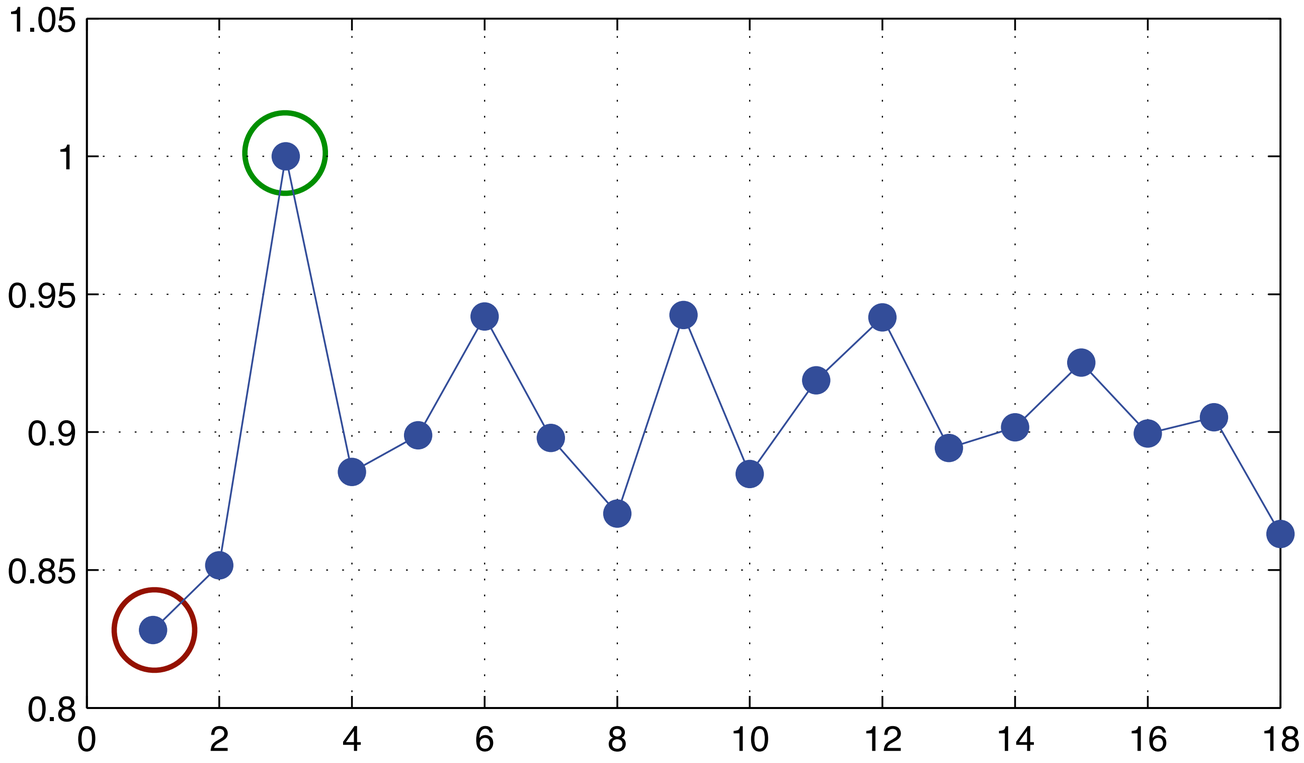
- For two unit-length vectors $\mathbf{x}$, $\mathbf{y}$

$$\mathbf{x}^\top \cdot \mathbf{y} = 1 - \varepsilon$$

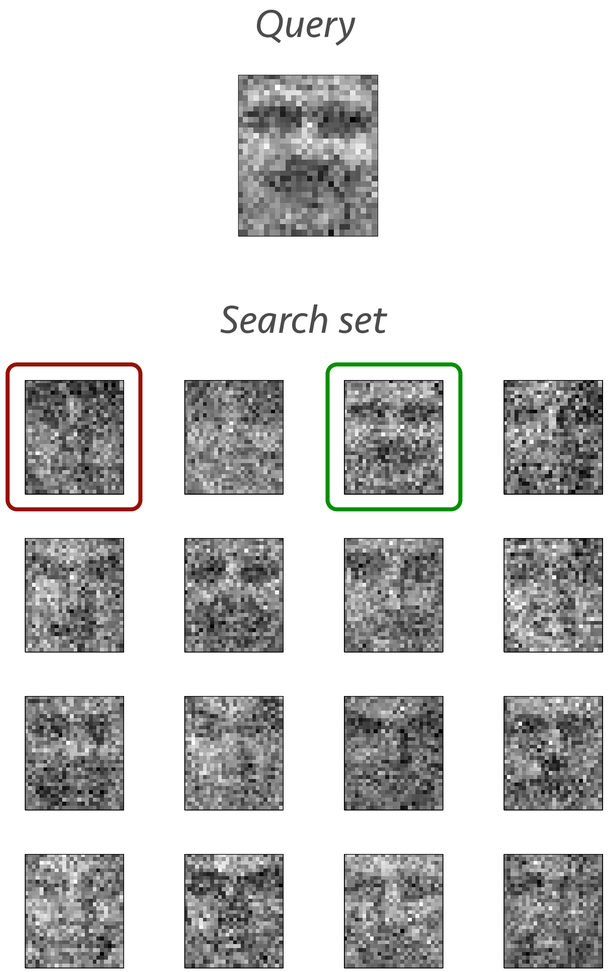
- When $\mathbf{x}$, $\mathbf{y}$ are increasingly similar, ε tends to zero

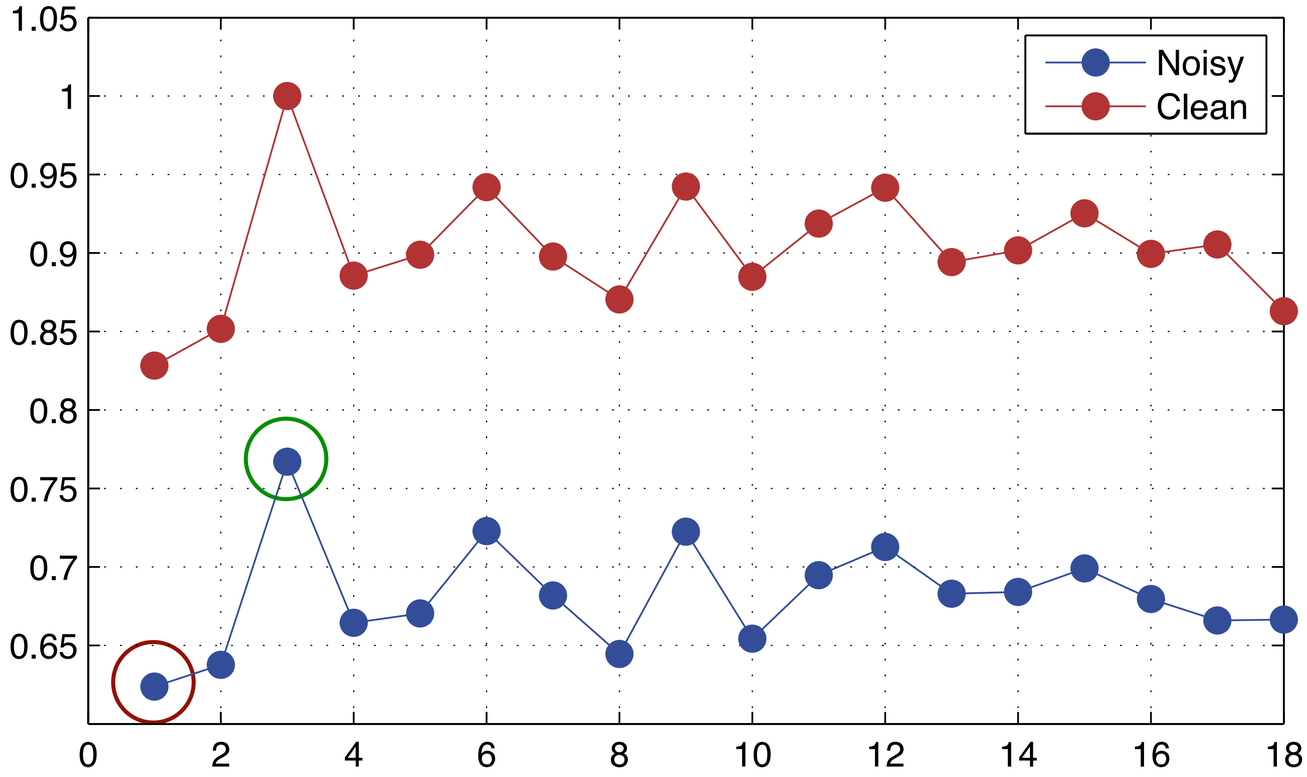
- If the dot product is maximized we have a close match

Interpreting the dot product



And it works ok for noisy inputs too



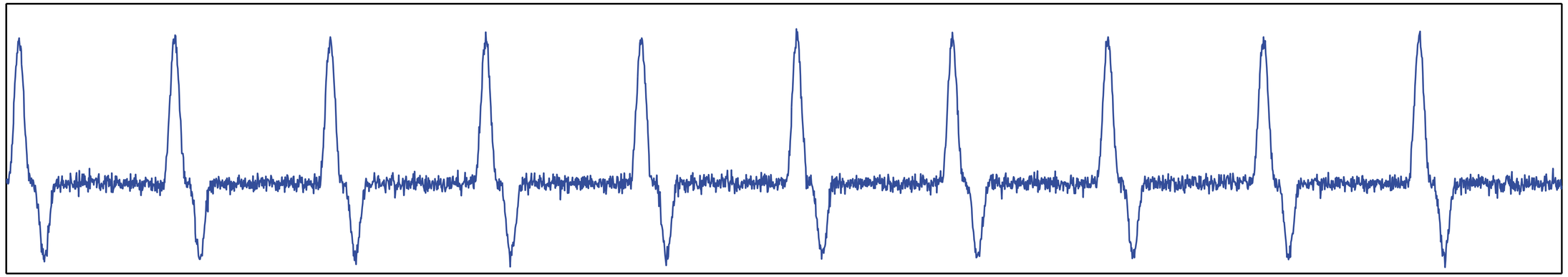


Template matching

- Our query is a template
  - Use dot products to compare it with inputs
- Simple and very fast detector/classifier
  - But it won't get you far as is!
- Let's signal-ize it

A tougher case

- "Heartbeat" data

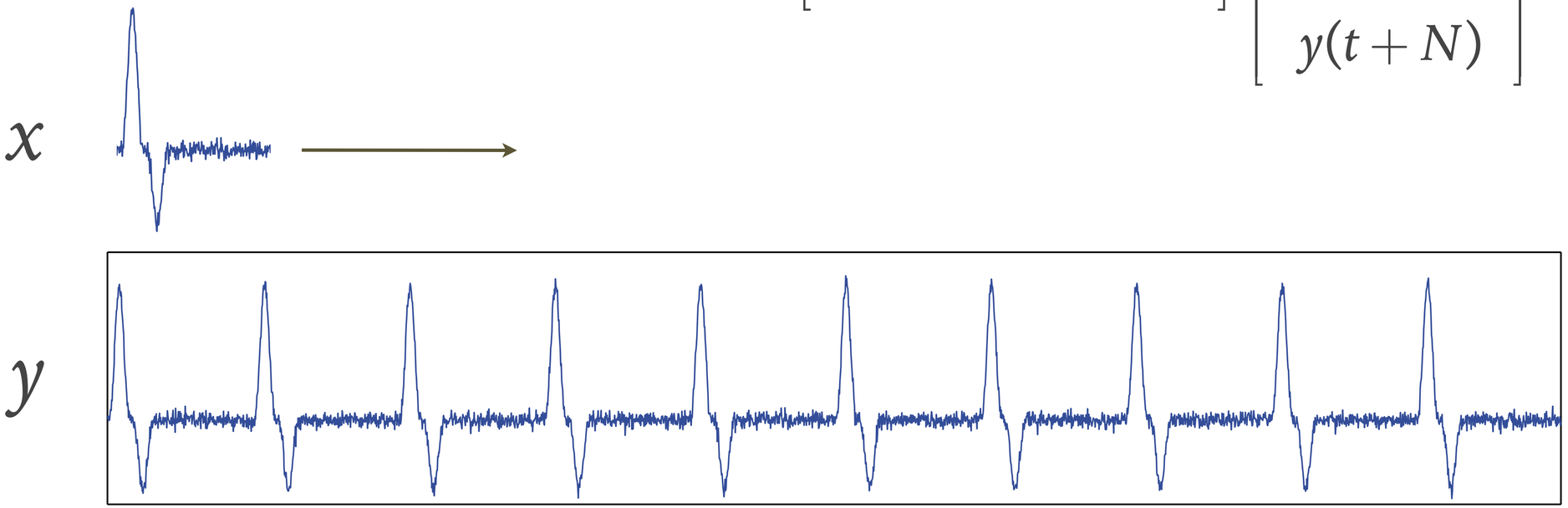


- How do we detect the beats?
  - They can be all over the place
  - How do we do the dot product?

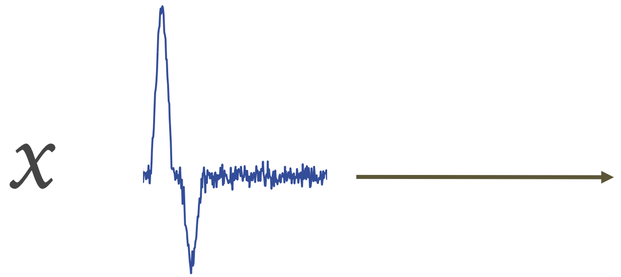
Brute force approach

- Do all possible dot products

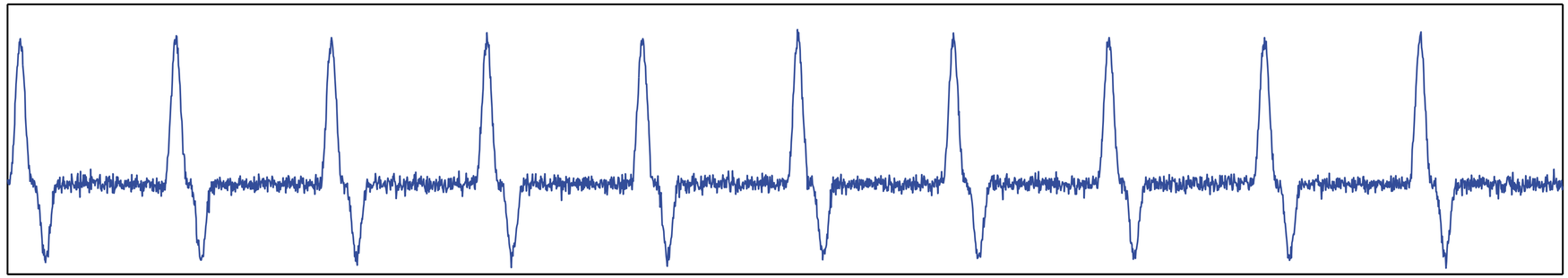
$$d(t) = \begin{bmatrix} x(1) & \cdots & x(N) \end{bmatrix} \cdot \begin{bmatrix} y(t+1) \\ \vdots \\ y(t+N) \end{bmatrix}$$

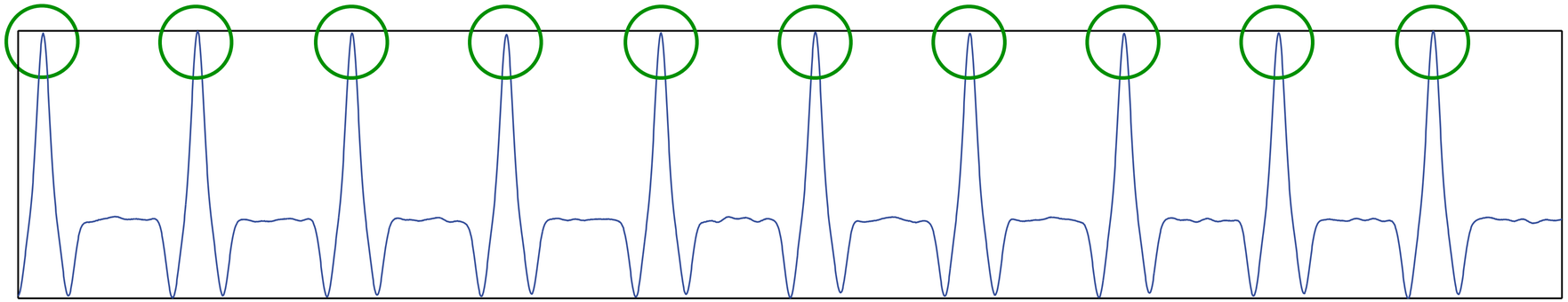


“Sliding” dot product



$$d(t) = \begin{bmatrix} x(1) & \cdots & x(N) \end{bmatrix} \cdot \begin{bmatrix} y(t+1) \\ \vdots \\ y(t+N) \end{bmatrix}$$





A closer look at this …

- Writing it as a summation it becomes a convolution
  - But with the template time-reversed

$$d(t) = \begin{bmatrix} x(1) & \cdots & x(N) \end{bmatrix} \cdot \begin{bmatrix} y(t+1) \\ \vdots \\ y(t+N) \end{bmatrix} =$$

$$= \sum_i x(i) y(t+i) = \overleftarrow{x} * y$$

$$\overleftarrow{x}(t) = x(-t)$$

Matched filter

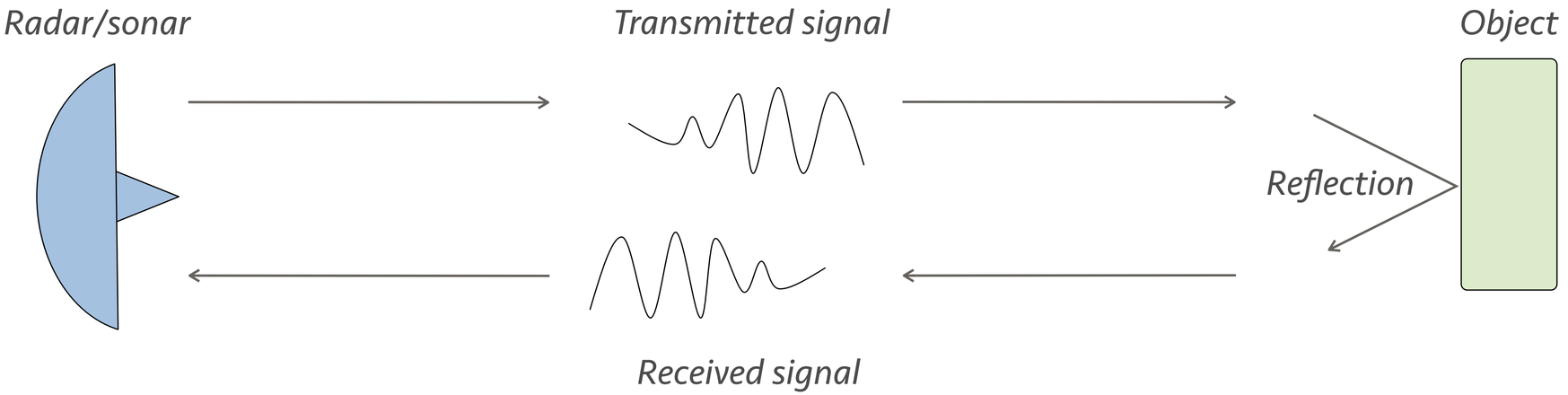
- This is known as a *matched filter*

$$d(t) = \sum_k x(k)\, y(t-k)$$

Template presence at time t — *Template (flipped in time)* — *Input*

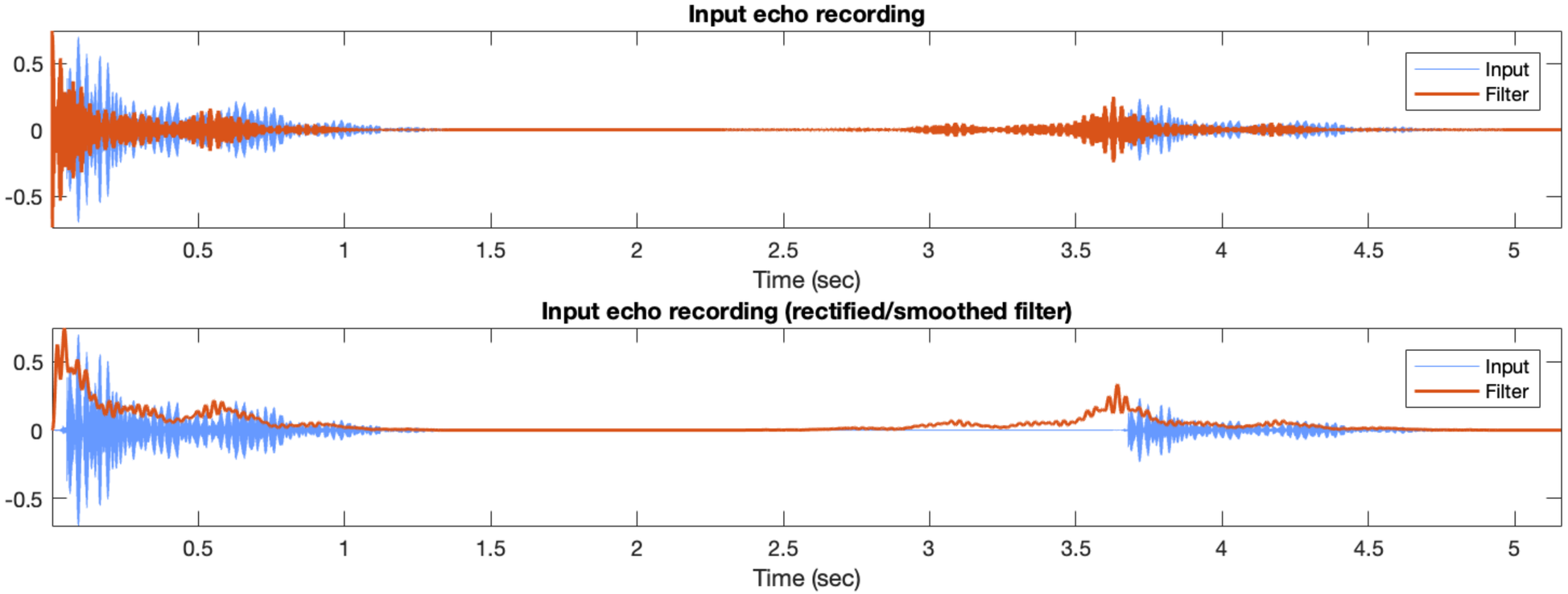
  - Why filter? Because we do a convolution
- Also known as *cross-correlation*
- Very common tool in communications, sonar, radar, neural networks, etc ...

Matched filters in sonar/radar

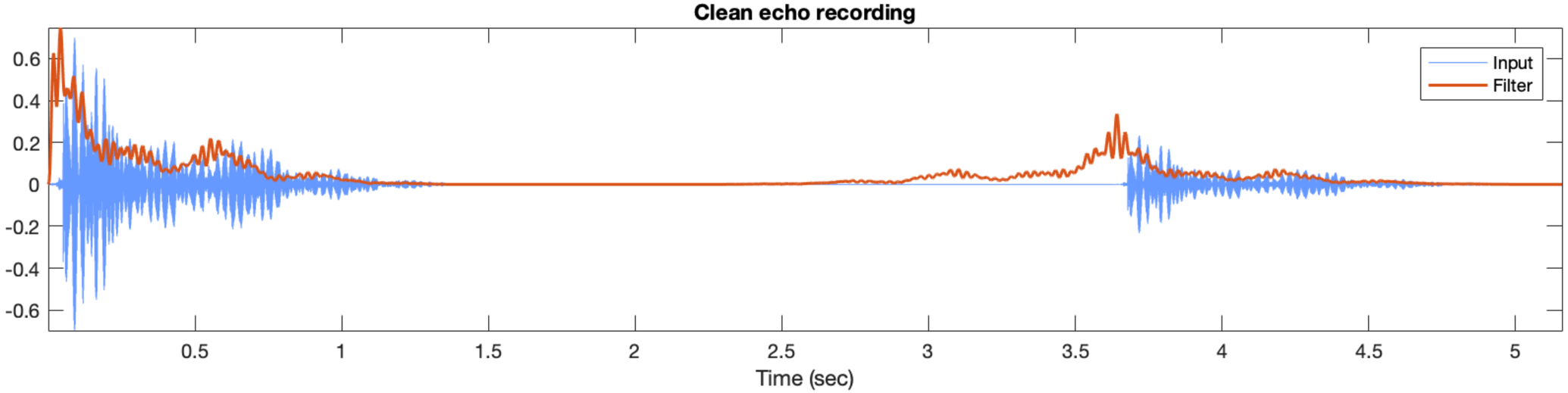


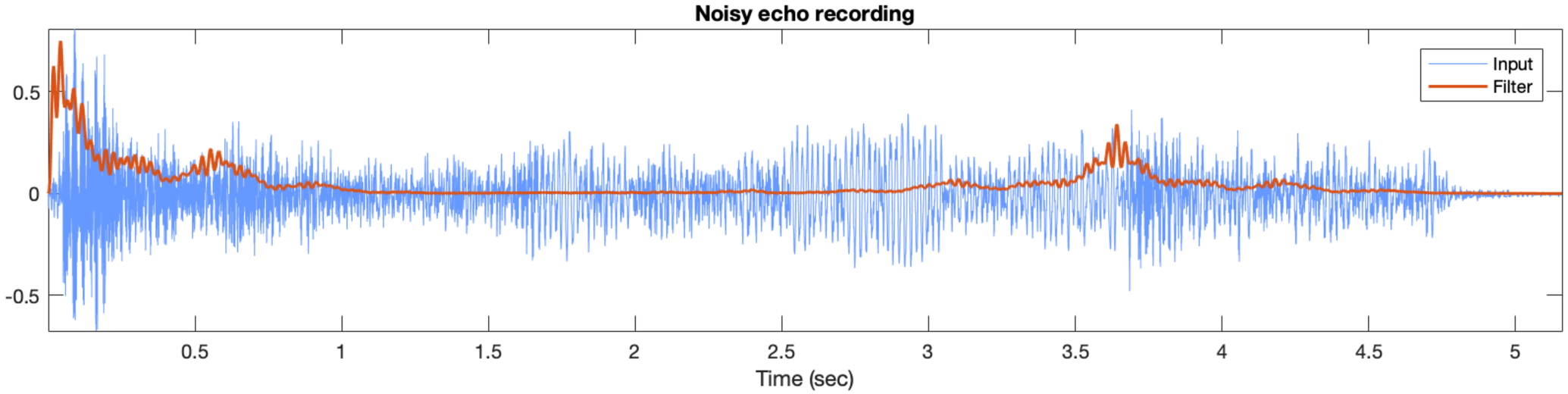
Some post-processing

- The matched filter output is a little messy
  - We mostly care about rough peaks, and its energy
    - So we rectify and lowpass filter



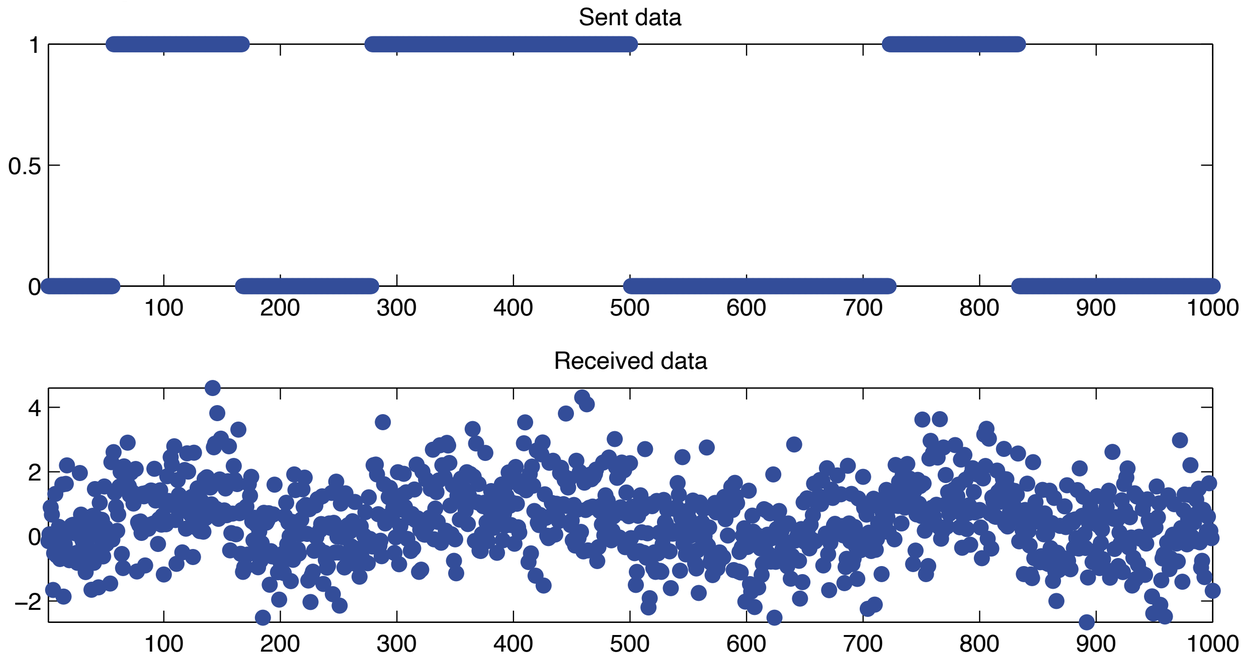
Most useful when with noise





Simple communications example

- You send 0/1's over the air
  - But you pick up some noise
- E.g.:



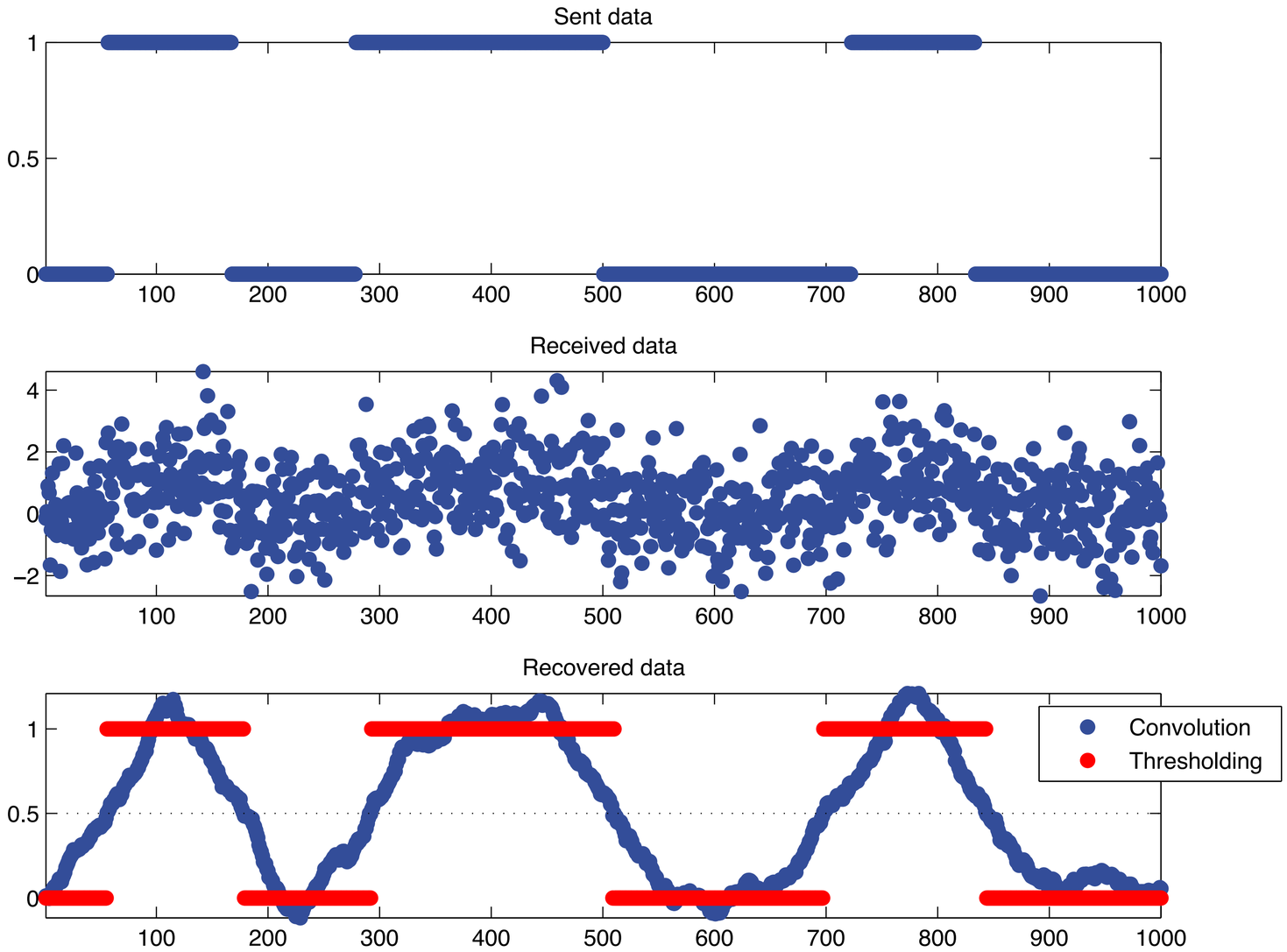
  - How do you recover the original data?

Detecting the 1's

- Make a template for them:
  - $\mathbf{x} = [1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1]$
  - This will detect a series of "1" symbols

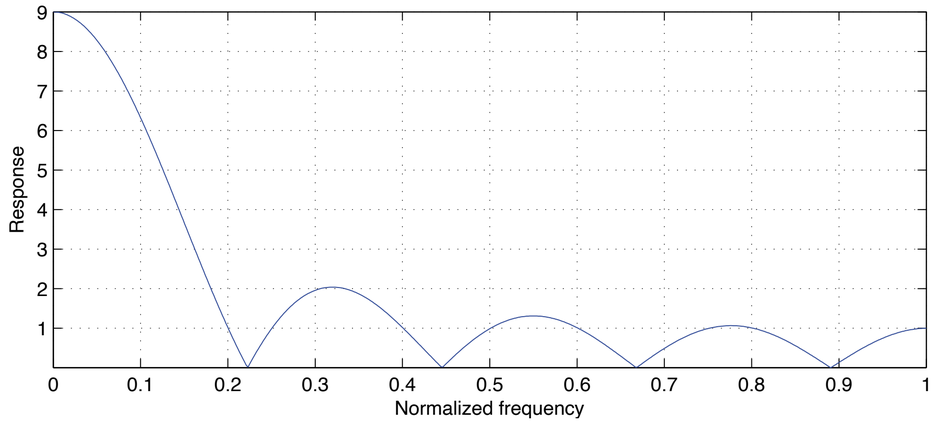
- Convolve template with the input
  - And measure when the result is above threshold
    - E.g. greater than 0.5
  - Where it is greater than 0.5 we have a series of "1"'s

Result


Sent data
1
0.5
0
100
200
300
400
500
600
700
800
900
1000
Received data
4
2
0
−2
100
200
300
400
500
600
700
800
900
1000
Recovered data
1
0.5
0
100
200
300
400
500
600
700
800
900
1000
Convolution
Thresholding

An interesting connection

- What is this filter?
  - $\mathbf{x} = [1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1]$



- It's a lowpass filter
  - We effectively remove the high frequencies (i.e. the noise)

- Two ways to interpret this operation
  - A template to match our signal
  - Or a denoising filter

Computational considerations

- The sliding dot product is slow
  - N multiplications/additions per input sample
    - No fun with large templates

- Fortunately we can use the Fast Fourier Transform!
  - Convolutions can be performed efficiently with the FFT

FFT-based matched filtering

- Convolution in the time domain maps to element-wise multiplication in the frequency domain

$$\mathbf{d} = \mathbf{x} * \mathbf{y} \Rightarrow \mathbf{F} \cdot \mathbf{d} = \Big(\mathbf{F} \cdot \mathbf{x}\Big) \odot \Big(\mathbf{F} \cdot \mathbf{y}\Big)$$

- But we have three sizes here: template, input, output
  - template is length N, input is M, output is $M+N-1$
  - What do we use?

Approach 1

- Zero pad everything to the largest size ($M+N-1$)
  - Ensures that we we have enough space to store the output
  - Doesn't change the result since we convolve with more zeros
    - Slightly redundant though

- Example: N = 1,000, M = 10,000,000
  - Time domain takes about 10 GFLOPs
  - FFT convolution takes about 200 MFLOPs

Approach 2

- Take small snippets of the input, convolve them with the template and add them together (convolution is linear!)
- How small?
  - Same size as template (N) to avoid excessive zero padding
  - But, the output will be $2N–1$, so we need to zero pad as much
- Example speedup, N = 1,000, M = 10,000,000:
  - Time domain: 10 GFLOPS
  - FFT: 200 MFLOPs
  - "Fast convolution”: 60 MFLOPs

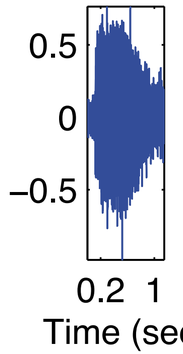
Matched filtering for event detection

- Let's try to detect gunshot sounds in a video

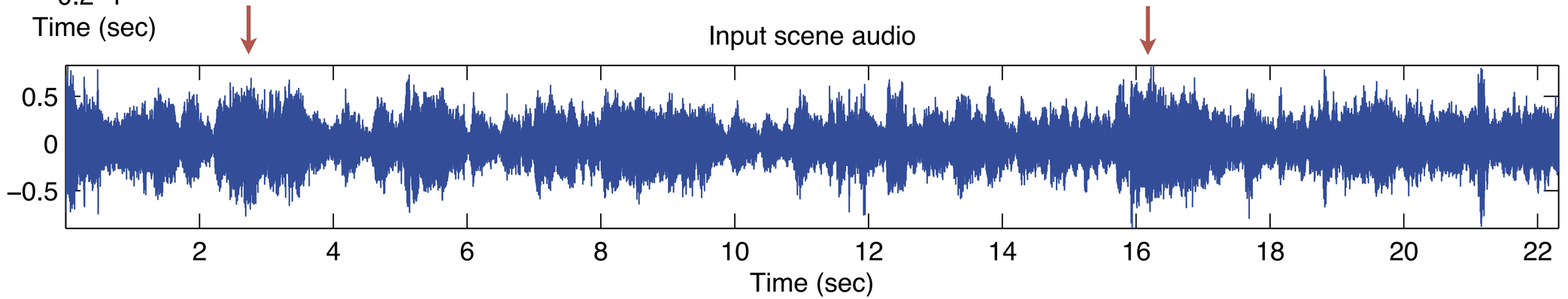
Input video

Extract audio only

Gunshot template

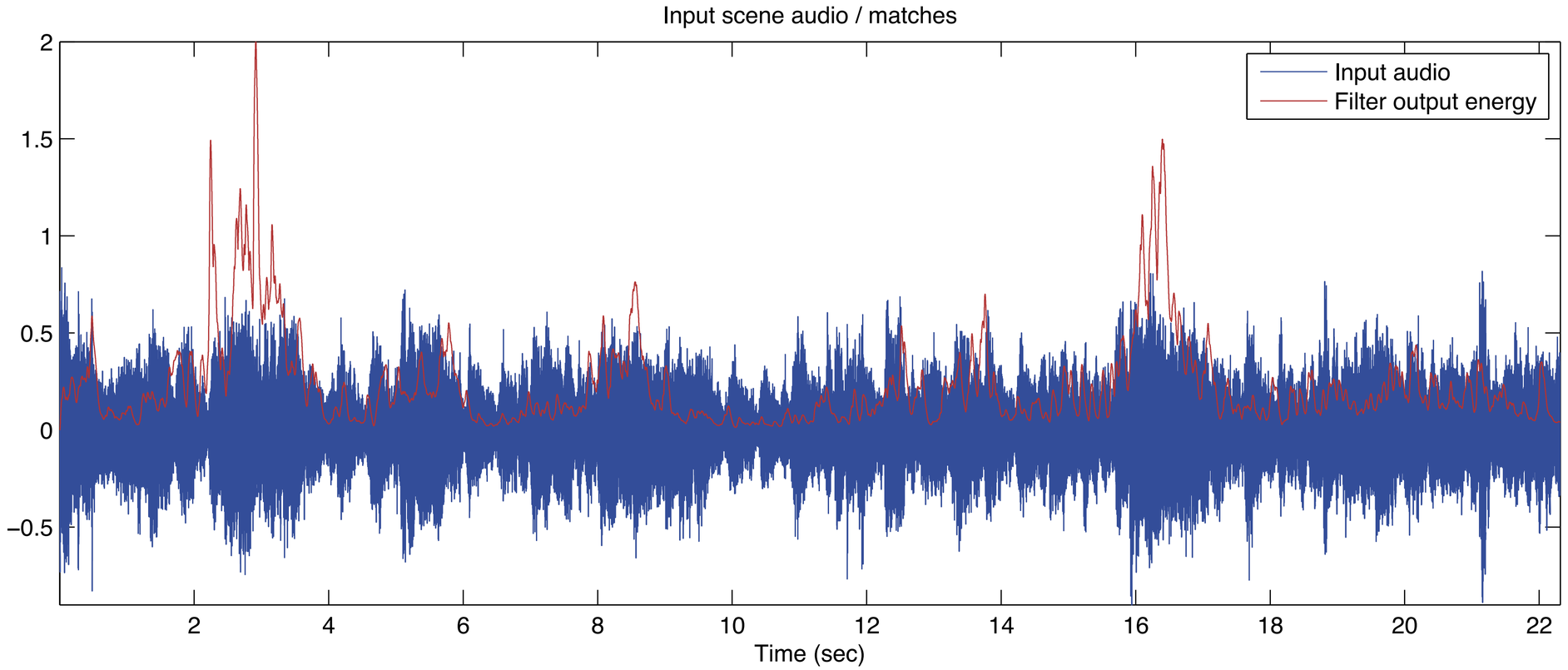


Input scene audio



Matched filter result

- Strong peaks where gunshots took place
  - Processing: Time domain: 1.8 sec, Fast Conv: 0.06 sec

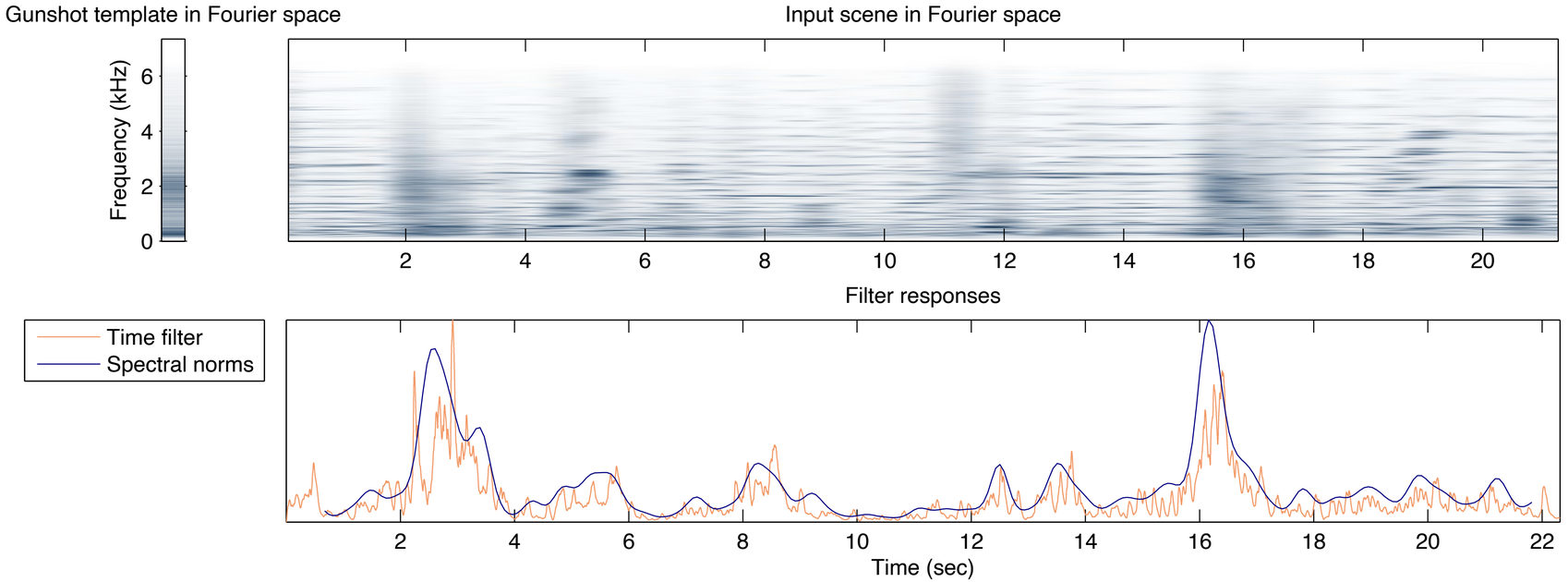


Going even faster

- The template and each time frame is a vector

$$\mathbf{t}_\tau * \mathbf{s}_\tau = \mathbf{F}^H \cdot \left[\left(\mathbf{F} \cdot \mathbf{t}_\tau \right) \odot \left(\mathbf{F} \cdot \mathbf{s}_\tau \right) \right] = \mathbf{F}^H \cdot \left(\mathbf{f}_t \odot \mathbf{f}_s \right)$$

$$\left\| \mathbf{t}_\tau * \mathbf{s}_\tau \right\|^2 = \left(\mathbf{F}^H \cdot \left(\mathbf{f}_t \odot \mathbf{f}_s \right) \right)^H \cdot \mathbf{F}^H \cdot \left(\mathbf{f}_t \odot \mathbf{f}_s \right) = \left(\mathbf{f}_t \odot \mathbf{f}_s \right)^H \cdot \left(\mathbf{f}_t \odot \mathbf{f}_s \right) = \left\| \mathbf{f}_t \odot \mathbf{f}_s \right\|^2$$



Generalizing to two dimensions

- In 1D the template was time-reversed
  - In 2D we flip both dimensions

$$d(i,j) = \sum_k \sum_l x(k,l)y(i+k, j+l)$$

- We now slide across both dimensions
  - E.g. left-to-right for all vertical offsets
  - FFT speedup still holds
    - But we now use 2D FFTs
    - Same caveats as before apply here as well (zero padding, etc.)

Digit recognition example

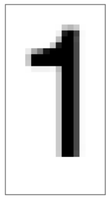
- Given the following input

Input

  - Find the digits by analyzing the image

Collect templates

- Create a set of templates for all digits

$\mathbf{x}_1 =$  $\mathbf{x}_2 =$  ...

- Filter flipped versions with input image
  - Each template will peak at digit's position

- Live demo!

A new problem

- What if we have a change in size?

Input

Broadening the search

- Resize the the templates to more sizes and run template matching again
  - How much resizing?
- This is N times more computation
  - When considering N scales
- Live demo

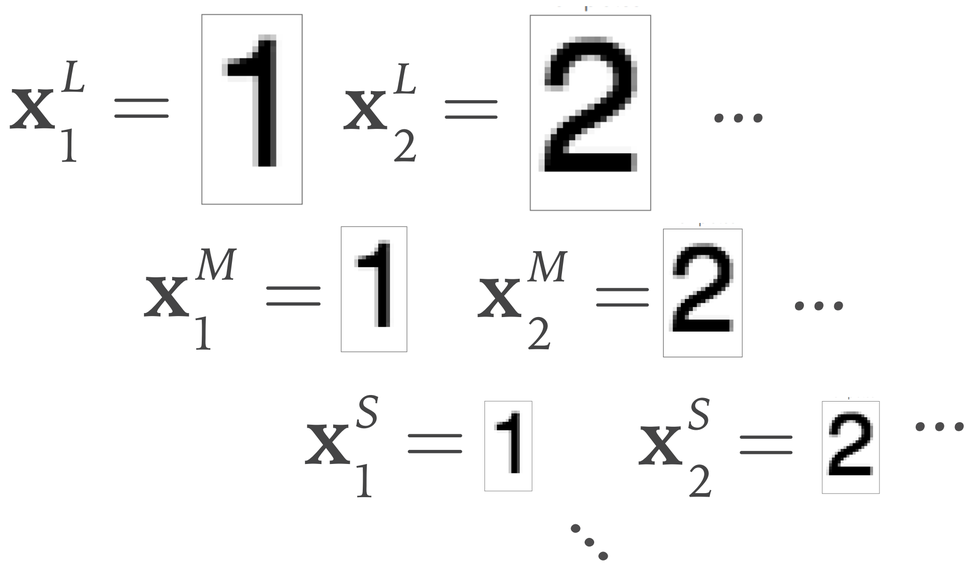
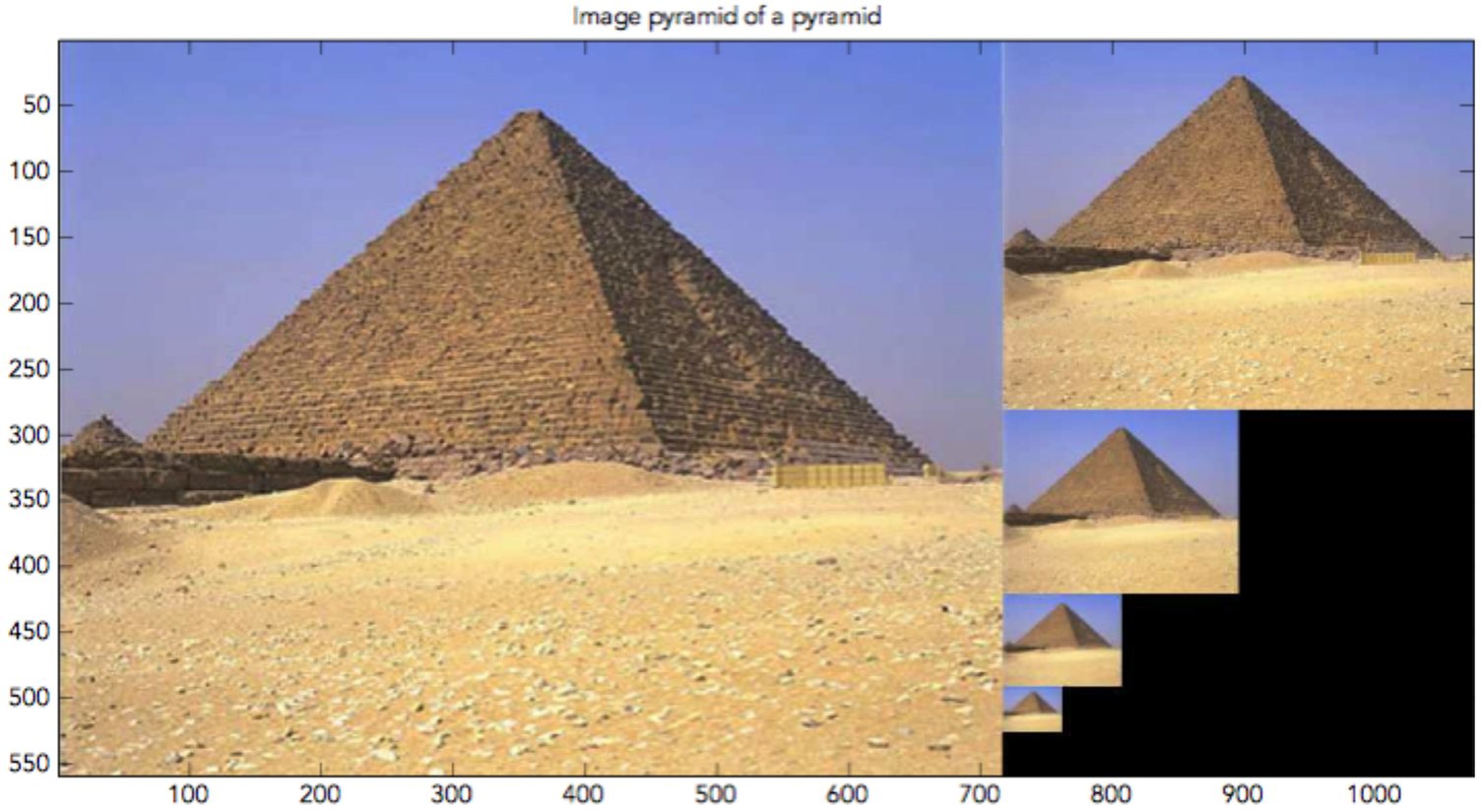


Image pyramids

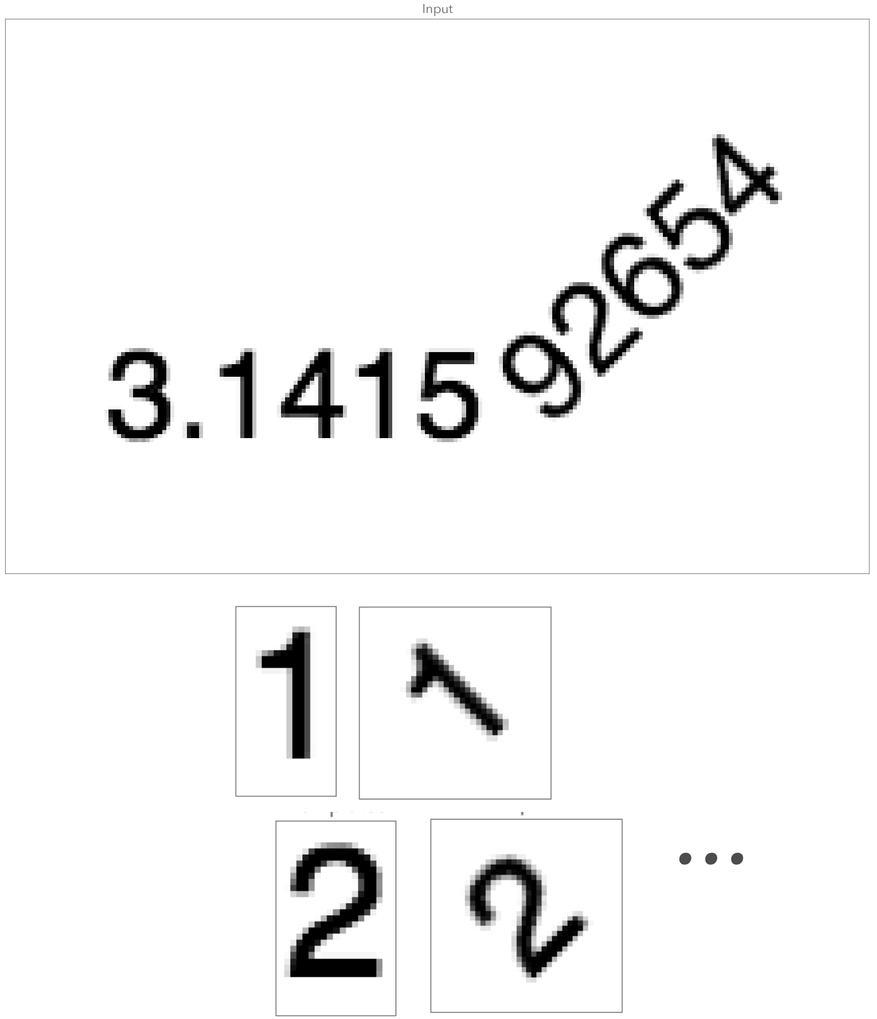
- Scaling the templates is not too efficient
  - Instead resample the input at multiple scales, not the template



Rotation?

- Same as before
  - Brute force search over all potential rotations

- One more multiplicative factor in computation

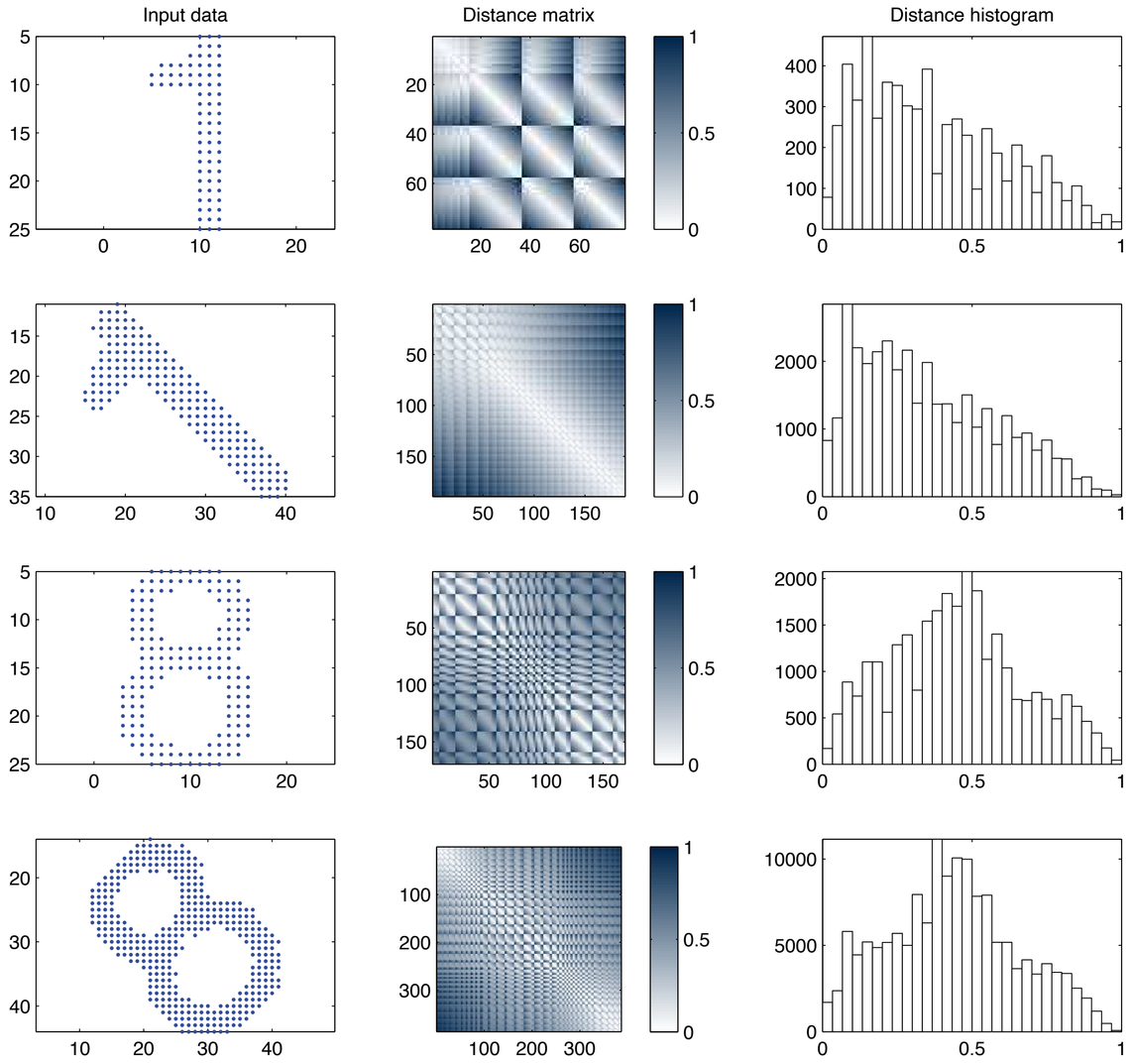


Some relief

- Rotation and scale and …, that's a lot of searching that needs to be done!
- There are some tricks we can pull
  - e.g. radial filters for rotation
- Better features
  - Choose, e.g., a rotation invariant feature
- Lots and lots of papers …
  - But some searching will be there

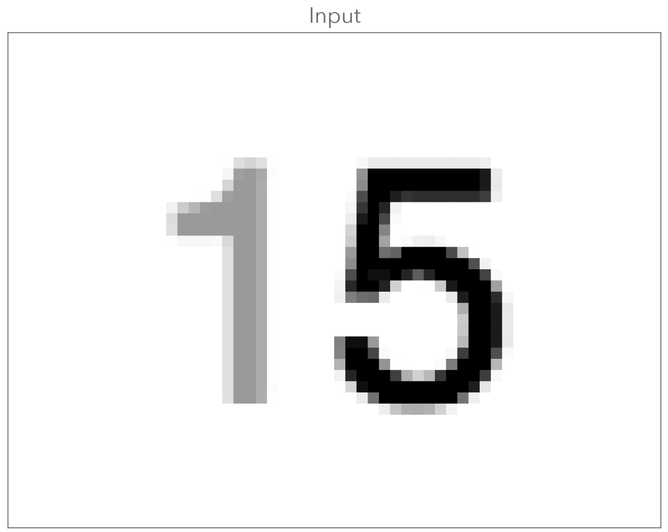
For example

- Treat significant pixels as points
  - Get their cross-distances
    - And normalize them
- The distances histogram is rotation- and scale- invariant
  - Do matching on that instead



One more problem …

- What happened to the normalization?!
  - Remember that vectors were unit-length?
  - We have not enforced that!
- Here's a potential problem case



Problems with scaling

- Regular cross-correlation is:

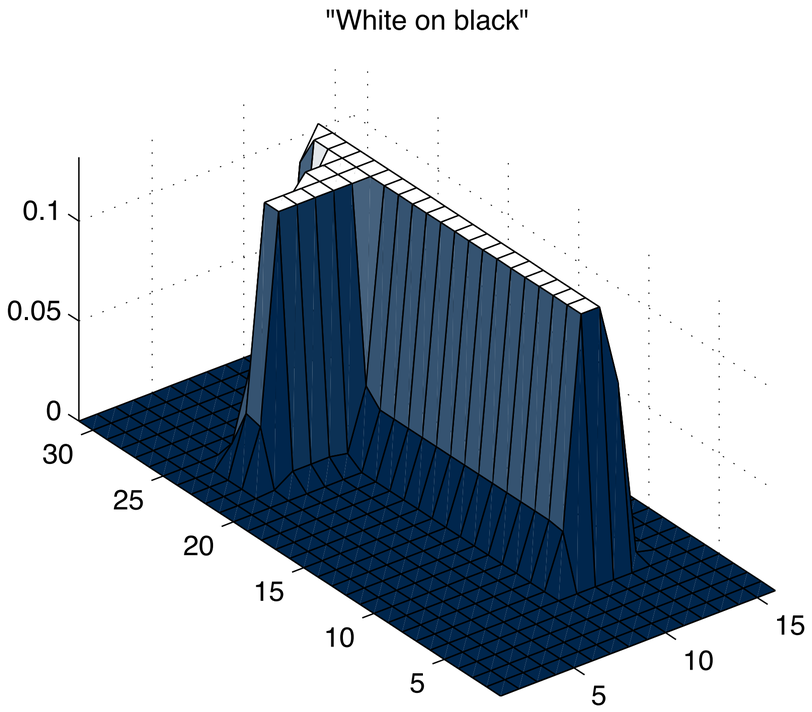
$$d(t) = \sum_k x(k)y(t-k)$$

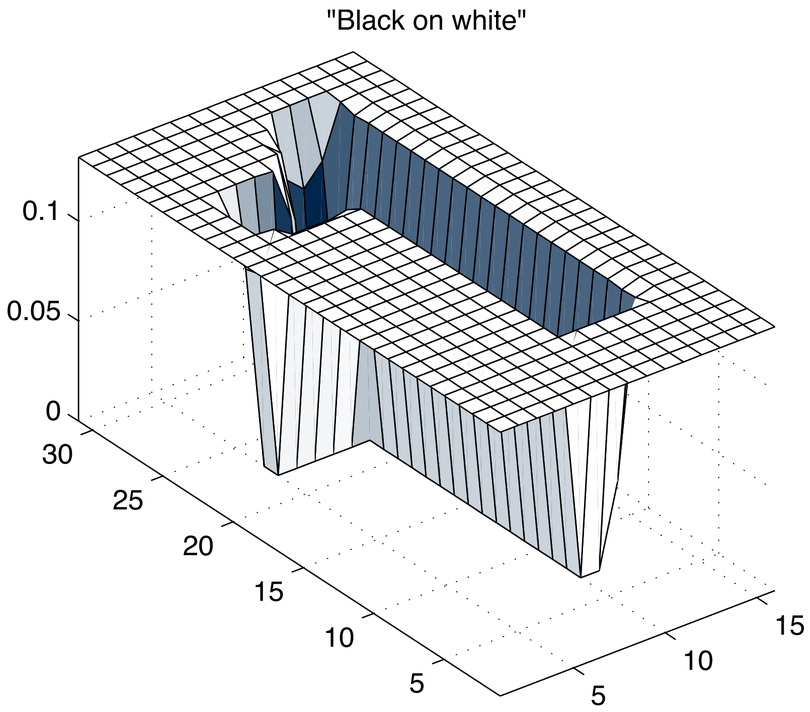
This is high, if this is high

- Output is highly dependent on the local magnitude of the input

In fact, I've been cheating all this time

- I used an inverted colormap (white:0, black:1)
  - I made sure that larger values were parts of interest
    - Do you see a new problem?





Solution 1: Normalize

- We really want:

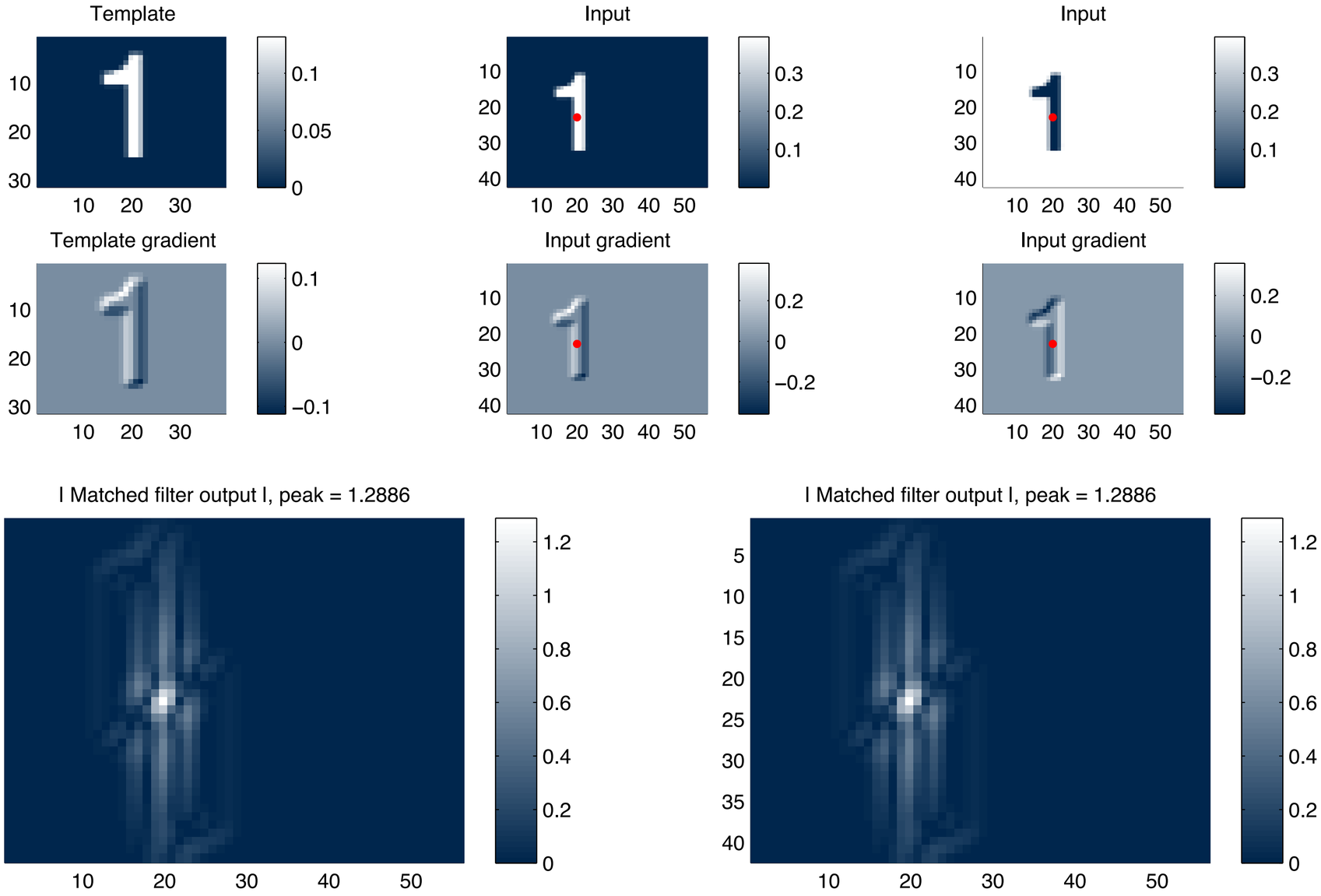
$$d(t) = \sum_k \frac{x(k)y(t-k)}{\sigma_x \sigma_{y(t)}}$$

  - Where the denominator ensures a unit-length dot product with the template
- Likewise we can generalize to 2-D, etc.
- But FFT speedup is gone now

Solution 2: Gradient filtering

- Detect what matters
  - Remember what your retina likes? (hint: edges)
- Perform detection on the gradient image
  - This abstracts the color and local intensity
- Much more robust detector (for images at least)

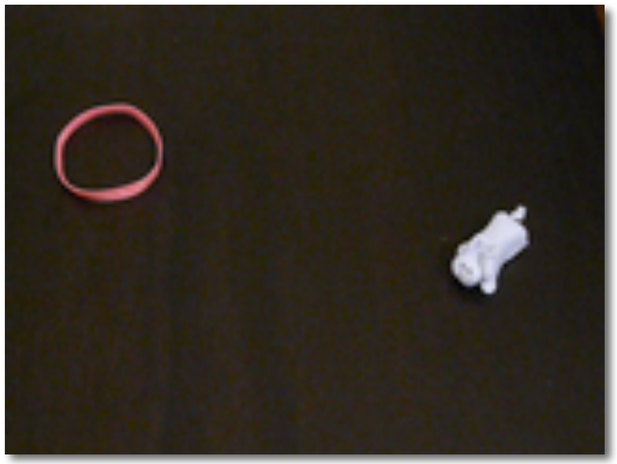
Example


Template
Input
Input
Template gradient
Input gradient
Input gradient
I Matched filter output I, peak = 1.2886
I Matched filter output I, peak = 1.2886

Movie example

- Track the ball movement
  - Convolve in 4D space

Input movie



120 × 160 × 3 × 138

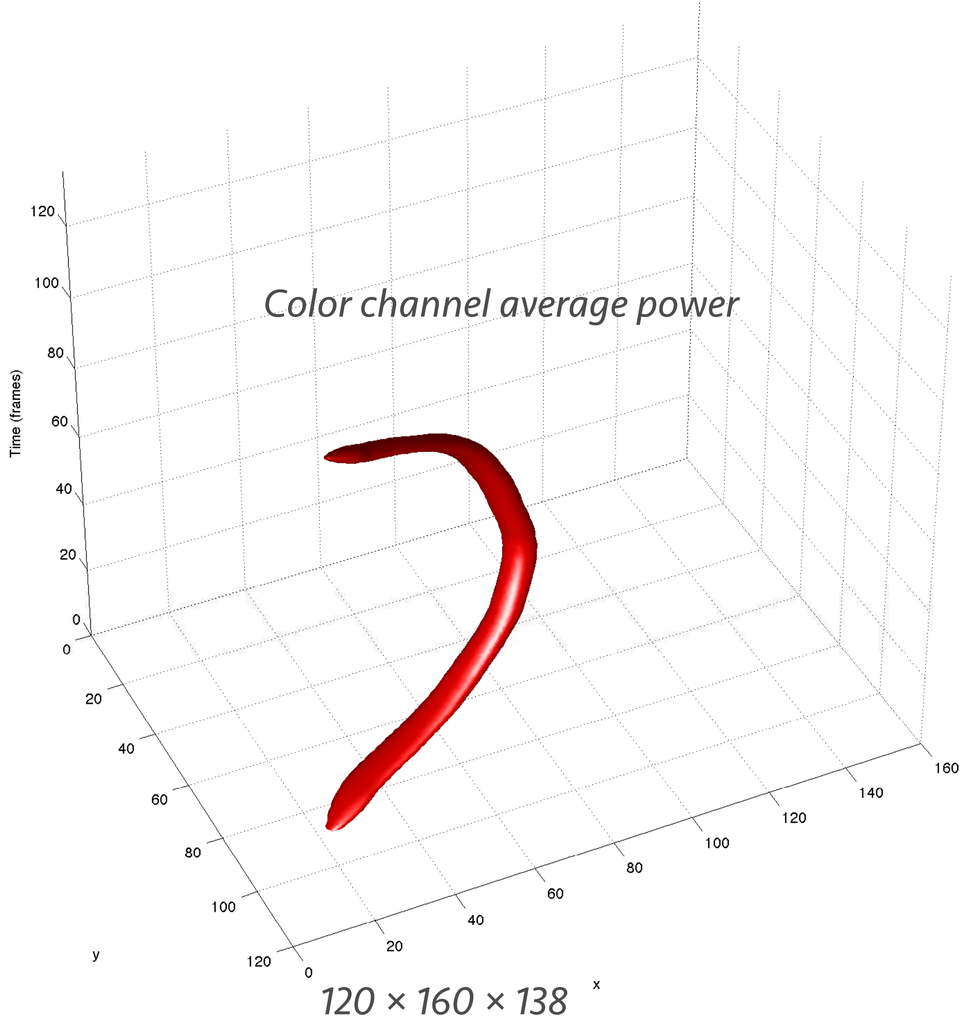
*

Ball template

28 × 29 × 3

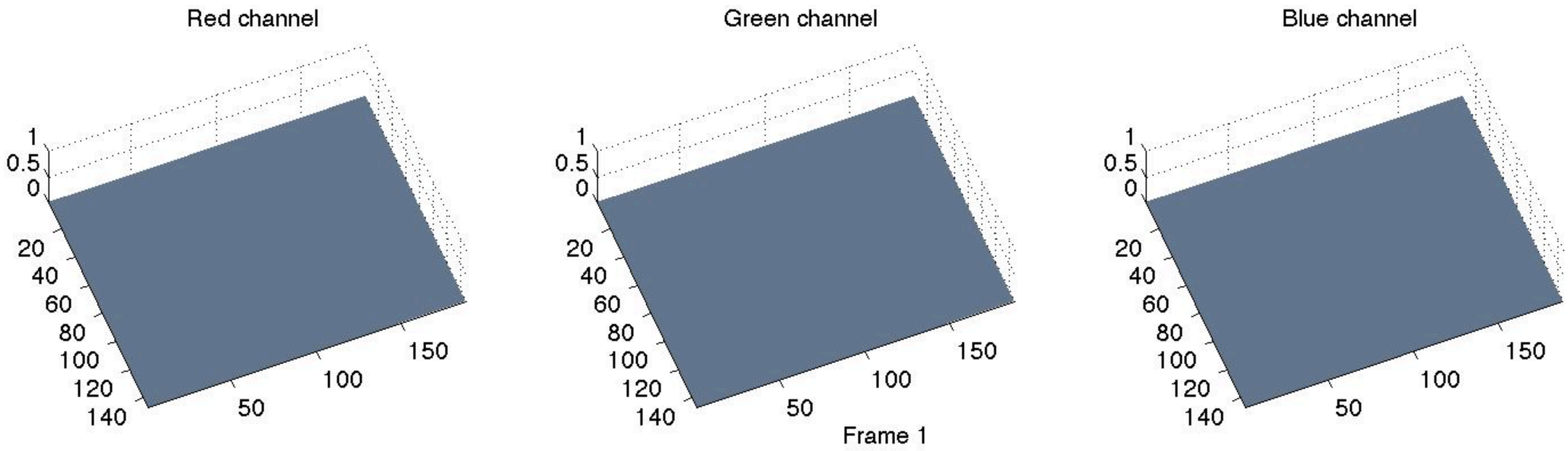
=

Color channel average power



120 × 160 × 138

Each channel separately



Template matching applications

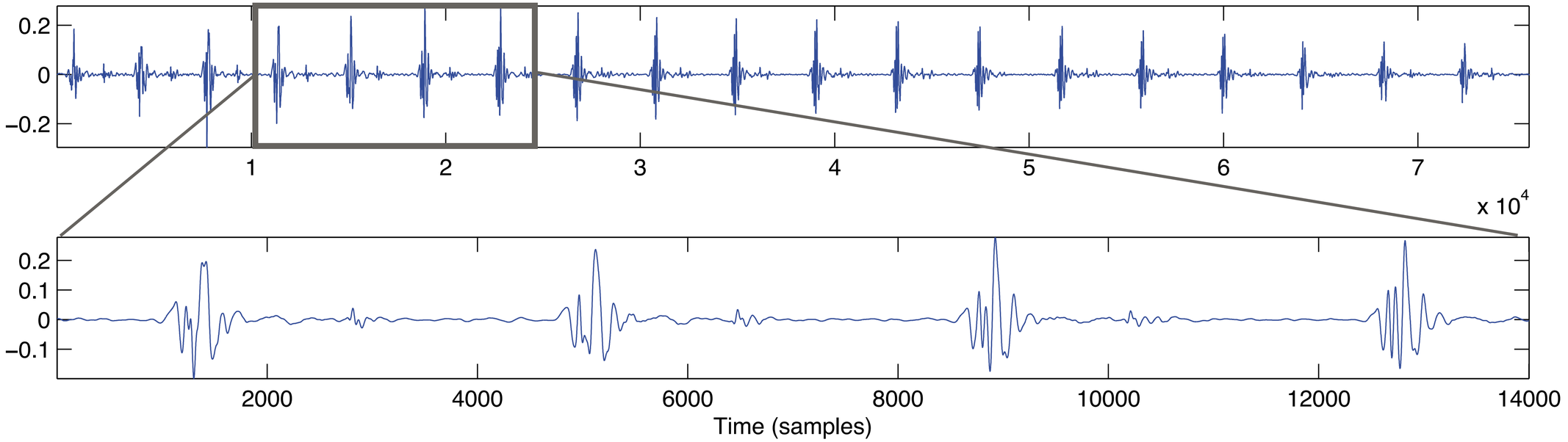
- Popular tool in computer vision
  - face/car/pedestrian/footballer detection
    - Templates are example images of the above
- Good for event detection in audio
  - gunshot detection, room measurements
- Bio/geological/space/underwater signals etc …

Matching without a template

- Sometimes we don't know the template
  - But we know it appears a lot
- We can use *auto-correlation* to find structure
  - Use the full input as a template
- Peaks will denote repeating elements

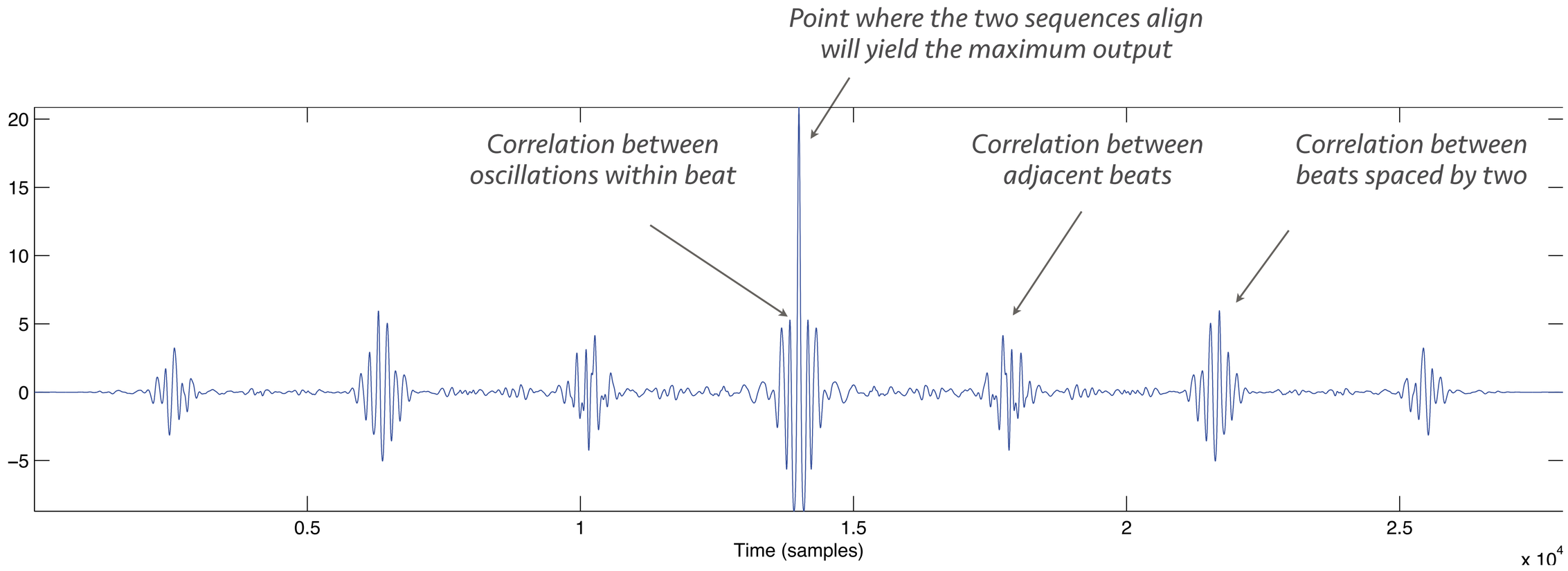
Back to the heartbeat example

- We don’t know the exact template, but there is a pattern that is repeating in time
  - We can still find structure regardless



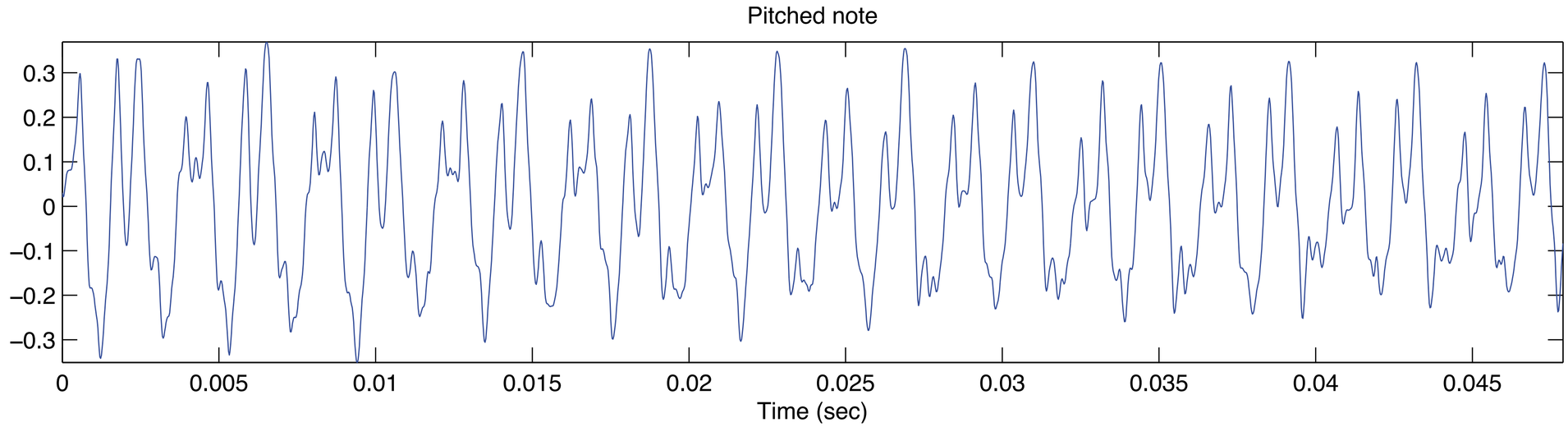
Autocorrelation

- Use the entire input as a template
  - Match the input on itself



A more complex case

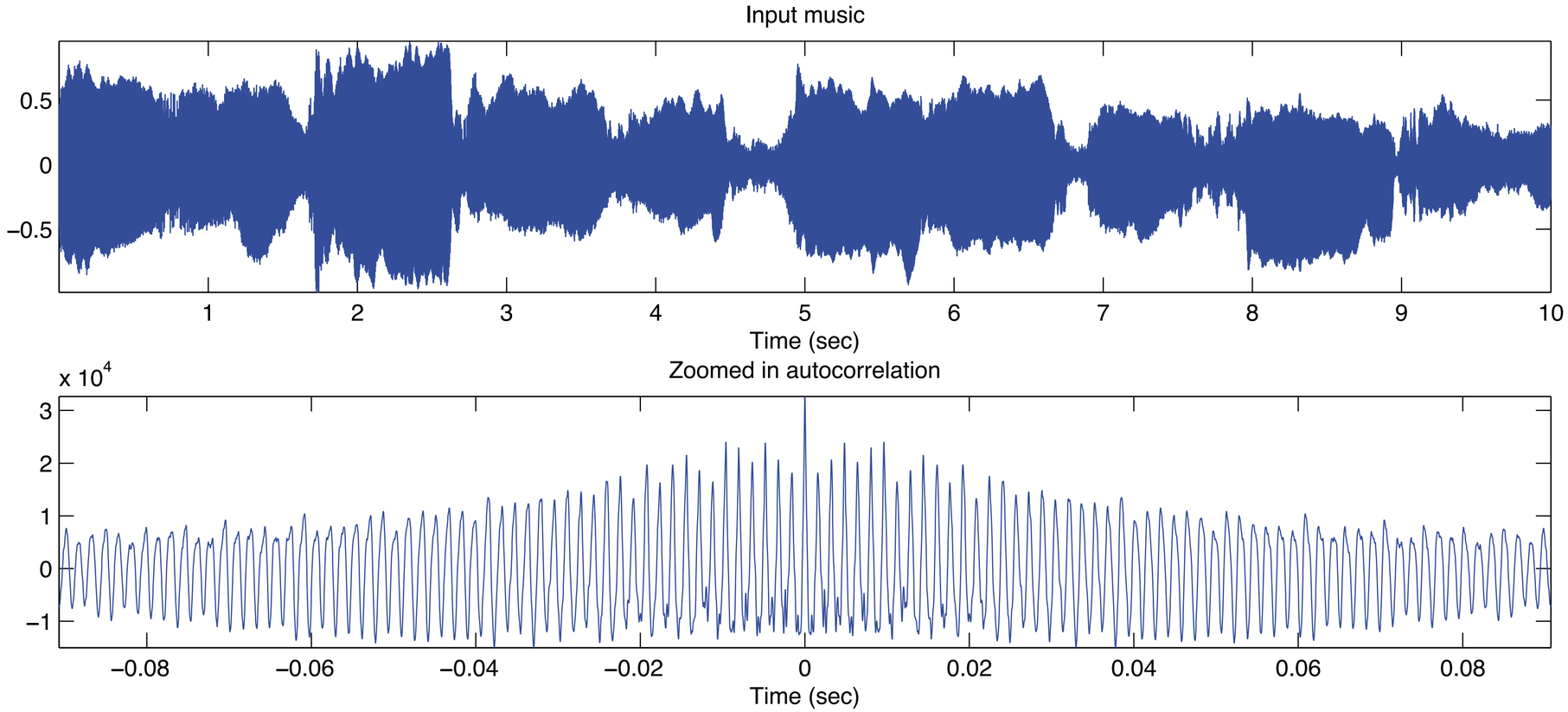
- Pitch tracking
  - Musical instruments tend to repeat a similar waveform, the rate of its repetition being the pitch (i.e. note played)



- But it can change over time!!

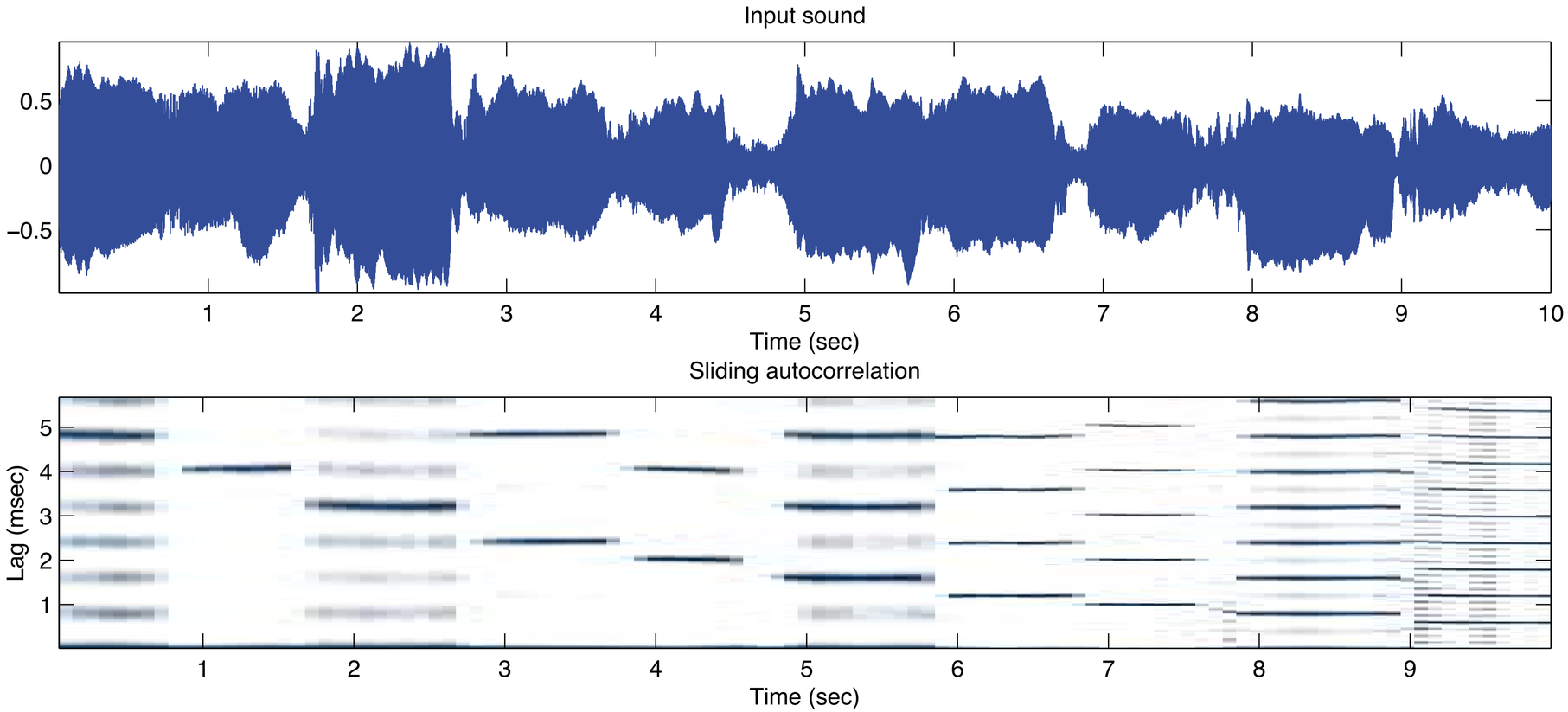
So we can't really autocorrelate

- Many peaks corresponding to various elements
  - No temporal information



Use localized autocorrelations instead

- For each analysis window perform an autocorrelation
  - Which you can also efficiently compute with the FFT



Probing matching criteria

- Revisiting the starting point
  - Suppose we instead tried to find the minimal distance between the template and the input

$$D(m,n) = \sum_i \sum_j \left| x(i-m, j-n) - y(i,j) \right|^2$$

    - This is a simple Euclidean distance

From Euclidean to dot product

- Expanding the square in the distance formula we get:

$$D(m,n) = \sum_i \sum_j \left| y(i,j) \right|^2 + \sum_i \sum_j \left| x(i,j) \right|^2 - 2 \sum_i \sum_j y(i,j) x(i-m, j-n)$$

- If we assume a locally constant input y
  - Minimizing D, maximizes the last term (which is the dot product!)
- Using the dot product we are ultimately minimizing a Euclidean distance between the template and the input

Getting to a likelihood

- The Euclidean distance implies a Gaussian

$$P(\mathbf{x};\mathbf{y}) \propto e^{-\frac{1}{2}(\mathbf{x}-\mathbf{y})^{\top}\cdot(\mathbf{x}-\mathbf{y})}$$

  - We can now (sort of) get a likelihood of match

- We can use various other metrics to define distance, each implying a different distribution
  - More on that later

And don’t forget features!

- We focused on detection on raw data
  - We can instead convolve on the feature weights
    - We sort of did that with the DFT and the gradients
- This can buy us noise robustness, invariance, and other desirable properties
  - You will rarely operate on raw data
- All of the previous theory applies here as well

Recap

- Detection of interesting elements
- Matched filtering
  - 1D to 2D
  - Scale/rotation invariance
  - Normalized cross-correlation
  - Detection on gradients
  - Autocorrelation
- From filters to likelihoods

Next lecture

- Linear classifiers
- Discriminant models
- Multi-class recognition

Reading

- Fast normalized correlation
  - http://scribblethink.org/Work/nvisionInterface/nip.pdf
- Transform invariant templates
  - http://www.springerlink.com/content/h345462721557808/
  - http://www.lps.usp.br/~hae/software/forapro/index.html

And we have a new problem set out!

- Less math, more coding!
- Important notes:
  - All submissions must be a rendered Python notebook
  - Write your own code for all numerical routines
    - i.e. PCA, ICA, NMF, etc. (do not use sklearn or equivalents)
- Due date is Oct 2nd (2 weeks from Friday)