# Filter Banks & Wavelets

- An issue with DFT is the non-locality.
  - ↳ exc: 2 frequencies alternating or one occurs completely first & then the other, would give us the same (similar) DFT.

## *1) Windowing: (Short Time Fourier Transform)

- Choose parts of the signal and perform a DFT on a window by moving the window throughout the signal.

$\Rightarrow \Delta x \Delta f \geq 1/2$ } Uncertainity principle

* ↳ If we take smaller windows, then the resolution in the fourier domain decreases & vice versa.

- We plot each DFT of each window, abs value and plot it using color to make a spectrogram. (White ⇒ Peaks, Black ⇒ None)
  (example in slides)

---

## *2) Haar Filter Bank: (Discrete Wavelet Transform) (Inverse)

**

- Another way to tackle the global nature of DFT.

⟶ Assume we are given a set of low pass & high pass filters. → Convolution.
ex: (Averaging → High pass, Derivative → Low pass)

Wavelet Procedure:
- We convolve signal with the filters, then perform down sampling. Reconstruction done with inverse filters & upsampling.

→ $y = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \end{pmatrix}$ Signal , $x * h = \frac{1}{2}\begin{pmatrix} x_0 - x_{-1} \\ x_1 - x_0 \\ x_2 - x_1 \\ x_3 - x_2 \\ \vdots \end{pmatrix}$ (High pass) , $h = [1 \;\; \underset{\uparrow}{1}] \cdot \frac{1}{2}$

$x * l = \frac{1}{2}\begin{pmatrix} x_0 + x_{-1} \\ x_1 + x_0 \\ x_2 + x_1 \\ x_3 + x_2 \\ \vdots \end{pmatrix}$ (Low pass) ; $l = [1 \;\; \underset{\uparrow}{1}] \cdot \frac{1}{2}$

→ We can see that $x$ can be achieved by using low & high pass results.

(Consider first row $\frac{1}{2}(x_0 - x_{-1}) + \frac{1}{2}(x_0 + x_{-1}) = x_0$

$\frac{1}{2}(x_0 + x_{-1}) - \frac{1}{2}(x_0 - x_{-1}) = x_{-1}$)

- So a single row in LP & HP can give us 2 values in $x$. We can use this to remove redundant features (Remove odd rows from $x*h$ & $x*l$, this is the down sampling part)

$$\Rightarrow D(x*h) = \frac{1}{2}\begin{pmatrix} x_0 - x_{-1} \\ x_2 - x_1 \\ \vdots \end{pmatrix}$$

$$D(x*l) = \frac{1}{2}\begin{pmatrix} x_0 + x_{-1} \\ x_2 + x_1 \\ \vdots \end{pmatrix}$$

- Once we have this, we can make changes etc since this is our transformed signal & we can reconstruct back from here.

⇒ First we upsample and then use the $l_s$ & $h_s$ reconstruction filters, to get back signal.

$$u(D(x*h)) = \frac{1}{2}\begin{pmatrix} x_0 - x_{-1} \\ 0 \\ x_2 - x_1 \\ 0 \end{pmatrix} = u(v_h)$$

$$u(D(x*l)) = \frac{1}{2}\begin{pmatrix} x_0 + x_{-1} \\ 0 \\ x_2 + x_1 \\ 0 \end{pmatrix} = u(v_l)$$

$\Rightarrow$ Now, using $l_2 = [1, 1]$, $h_2 = [1, +1]$

$$u(v_l) * l_2 = \frac{1}{2}\begin{pmatrix} x_0 + x_{-1} \\ x_2 + x_1 \\ x_2 + x_1 \\ x_0 + x_{-1} \end{pmatrix}, \quad u(v_h) * h_2 = \frac{1}{2}\begin{pmatrix} x_0 - x_{-1} \\ x_2 - x_1 \\ x_2 - x_1 \\ x_0 - x_{-1} \end{pmatrix}$$

- Now on adding them up we get the original

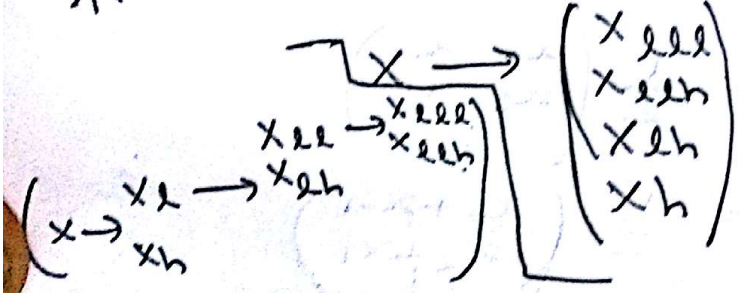
$$x = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \end{pmatrix}$$

- This part is the inverse wavelet transform
- Hence we can transform & get back as well.

---

- We can apply this in multiple stages,

$\Rightarrow$ Multi stage filter bank (example in slides)




- If we only reconstruct from $x_{lll}$, then we have a compression of 8-fold.

(We will start seeing artefacts in the reconstructed signal though)

---

**Note:** In the transform we require a set of 4 filters, Haar proposed the averaging & derivative filters, but we could use any set of filters for this.

⇒ For finite signals, we can either pad with zeroes or repeat signal on each side, perform filtering on extended signal & truncate back to required length.

* ↳ Discrete Wavelet Transform (DWT)
(Details in slides)

---

## * Matrix Representation of Wavelet Transform
(Discrete Wavelet Transform)

*

• Consider $x = [a, b, c, d]$ (Convolution, flip filter!) (Check all examples)

a b c d a b c d a b c d

$\Rightarrow x_l = \frac{1}{2}\begin{pmatrix} a+d \\ b+a \\ c+b \\ d+c \\ \vdots \end{pmatrix}$, $x_h = \frac{1}{2}\begin{pmatrix} a-d \\ b-a \\ c-b \\ d-c \\ \vdots \end{pmatrix}$

$l = [1\ 1] \cdot \frac{1}{2}$ (↑)

$h = [-1\ 1] \cdot \frac{1}{2}$ (↑)

• Downsampling, $D(x_l) = \frac{1}{2}\begin{pmatrix} a+d \\ c+b \end{pmatrix}$

$D(x_h) = \frac{1}{2}\begin{pmatrix} a-d \\ c-b \end{pmatrix}$

$X = \frac{1}{2}(a+d, c+b, a-d, c-b)$

↓

Represent as a matrix

$W_4^a = \frac{1}{2}\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & -1 \\ 0 & -1 & 1 & 0 \end{pmatrix}$, $X = W_4^a x$ (DOT)

$x = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$

~~Upsampling~~ ⇒ Work on representations, compress, modify etc.

• Suppose after changes $X' = [A\ B\ C\ D]$

• Upsampling, $u(V_l) = \begin{bmatrix} A \\ 0 \\ B \\ 0 \end{bmatrix}$, $u(V_h) = \begin{bmatrix} C \\ 0 \\ D \\ 0 \end{bmatrix}$

⇒ Now we would convolve with the l's & h's filters.

$\Rightarrow u(x_l) * l_s = \begin{bmatrix} A \\ B \\ B \\ A \end{bmatrix}$ , $l_s = [1 \quad 1]$ ↑

$u(x_h) * h_s = \begin{bmatrix} C \\ -D \\ D \\ -C \end{bmatrix}$ , $h_s = [-1 \quad +1]$ ↑

(Convolution, flip filter)

$\Rightarrow$ Now on adding the two up we have our original signal.

$$x = (A+C,\ B-D,\ B+D,\ A-C)$$

$\Rightarrow$ Matrix view, $x = W_4^s X$ (IDWT)

$$W_4^s = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \end{pmatrix}$$

- We can see that $W_4^s \cdot W_4^a = I$ (constant) ↗

* $\Rightarrow$ Also both the matrices are orthogonal rowwise. ≡ The rows together also
* form some kind of basis.

—x—

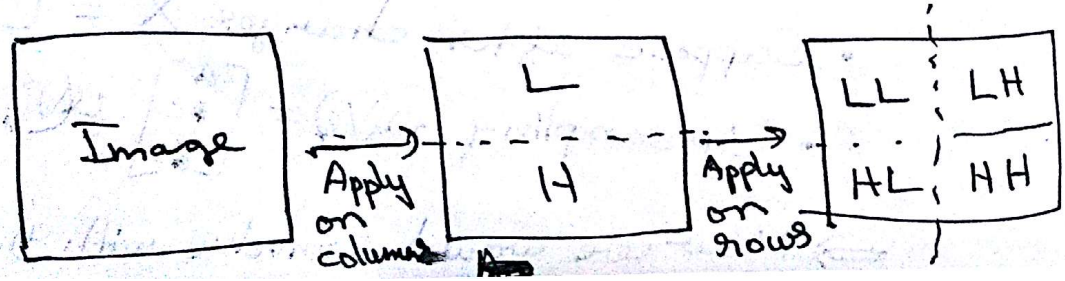
22

## * Wavelet Transform in 2D :

* • Similar to 2D DFT.

$\Rightarrow$ Apply on columns first and then apply on the rows. (Or vice versa)

$W_a^l(A) =$



- We can now repeat this process over any of the 4 parts & so on....

Note: Curvelets

---

## Applications of Wavelet Transforms:

*1) **Edge Detection:** (examples in slides)

- Ignoring the LL part & reconstructing from LH, HL & HH gives us the edges. → Horizontal, Vertical, Diagonal.

⇒ We can perform the transform multiple times on the LL part, so that we could zero out only smallest LL part & then reconstruct.

2) **Compression:** (JPEG 2000)

- These are useful for compression since storage of LH, HL & HH bands are quite redundant → they repeat a lot.

⇒ So these bands are encoded better to achieve the highest ~~[redacted]~~ reduction in size.

3) **Watermarking:**

- Carefully add watermark details within bands LH, HL, HH, LL so that

they are hard to remove and people cant just cut/crop them out.

4)

5) Denoising: (Very effective)

- We will not play around with LL since it is already a filtered low pass image.

⇒ Now in the others we apply a thresholding

  - Hard: If above threshold → 1 else 0
  - Soft: If above threshold → subtract threshold else 0.

- We now reconstruct back & the noise is gone.

———x———

## Multi Scale Wavelet Transform: (1D)

(Slides for details)

- We keep repeating the transform on the LL (in 2D) or L part in 1D leaving H unchanged.

? ex: [9 7 3 5 6 10 2 6] → [6; 0; 2 2; 1 -1; -2 -2]

$L_0$ $S_0$ $S_1$ $S_2$

$\alpha_0$ = Base image & $\delta_0, \delta_1, \delta_2$ are details at each layer.

⇒ The matrix form $\begin{bmatrix} 6 \\ 0 \\ 2 \\ 2 \\ 1 \\ -1 \\ -2 \\ -2 \end{bmatrix} = A \cdot \begin{bmatrix} 9 \\ 7 \\ 3 \\ 5 \\ 6 \\ 10 \\ 2 \\ 6 \end{bmatrix}$

$(A = A_0 \cdot A_1 \cdot A_2 \cdots)$

↑ This decomposition can be done (Slides)

- A could be derived by multiplication of matrices at each level and it can be derived from $\psi(x)$.

⇒ $\psi(x) = \begin{cases} 1 & , 0 \le x \le 1/2 \\ -1 & , \frac{1}{2} < x \le 1 \\ 0 & , \text{otherwise} \end{cases}$ } Father function

$\psi_{jk}(x) = \psi(2^j x - k)$ → Its like scaling in space as well as time.

↑ Time ↑ Space

$, j \ge 0$

$0 \le k \le 2^j - 1$

- Using $\psi_{jk}$ we can create our matrix A. (Each row is orthogonal, so they form a basis)

⇒ This is specially for Haar Wavelets.

- Since rows form a basis, where we keep repeating till $\alpha_0$ is a single element.

## * Image Compression:

- Data is used to store information

⇒ Redundancies → Encoding information
→ Spatial & Temporal redundancies
→ Irrelevent information. (Perceptual redundancies)

- We exploit these to use them for compression.

⇒ Y, Cb, Cr experiment (Slides for example) → We are highly sensitive to the Y channel but not Cb, Cr.

*

⇒ Image comparision → SNR } Signal to noise ratio
→ PSNR } Peak signal to noise ratio.

- Best way to evaluate is to use people to manually judge.

$\left(10 \log_{10}\left(\frac{255^2}{E_{ms}}\right)\right)$

$E_{ms}$ → Similar to variance.

———x———

## * Self Energy:

- Something that is rare (low probability) has more self energy.

$I = -\log_2 p$ } $p$ = probability of occurance

## * Entropy:

- $H = -\sum_{i=0}^{L-1} p_i \log_2 p_i$ } Mean of self energies.

## * Shannon's Theorem: (Memoryless, independent probabilities)

↳ If not then more compression can be done

- He stated that the minimum number of bits required to store information about one pixel is equal to the entropy of the source. (For lossless compression)

(Anything using a higher bitrate has redundancies).

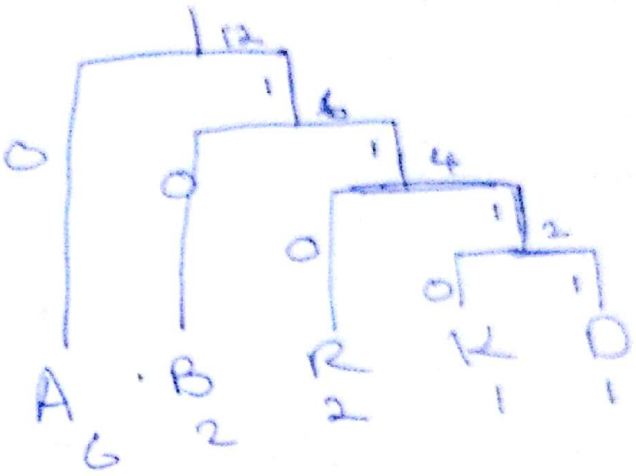
———x———

# *Huffman Coding: (Lossless compression) (Prefix free)

1) Based on probabilities of each value.

ex: ABRAAKADABRAA

A — 6
B — 2
R — 2
K — 1
D — 1




∴ A ⇒ 0
B ⇒ 10
R ⇒ 110
K ⇒ 1110
D ⇒ 1111

Bits required:
• 6x1 + 2x2 + 2x3 + 1x4 + 1x4 = 24 bits

(We can also end up getting other encodings but generally the bits required are the same)

• In this manner we give small encodings to the most occuring symbols and vice versa.

---

# *2) Run Length Coding: (Lossless compression) (OSHA rules)

000001111100 → 14 bits
(5, 6, 3)

⇒ We can just store [5, 6, 3] and I assume I always start with zeros.

ex: 11000001111 ⇒ [0, 2, 5, 5]

---

## *3) Arithmetic Coding: (Lossless compression) (OSAA notes)

**

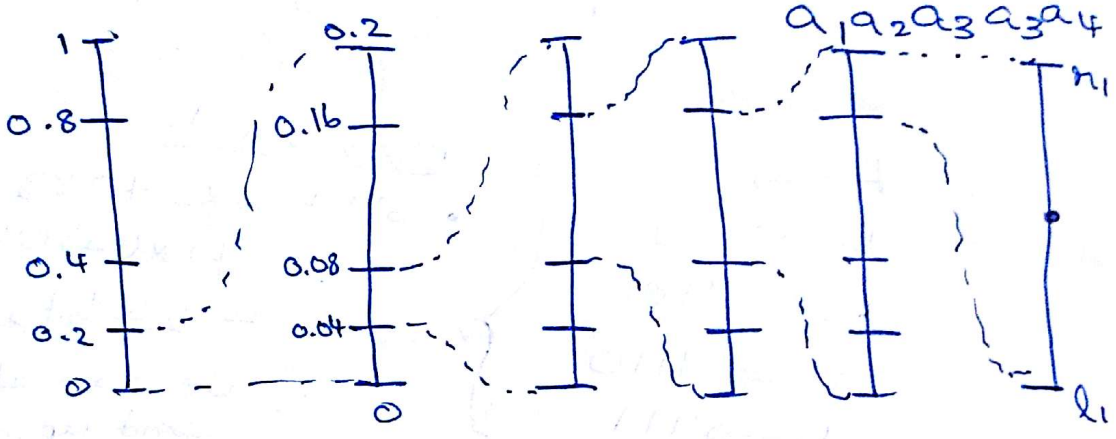
- Based on probabilities on each symbol, divide current window as required.

⇒ Entire message encoded by a single number.

ex: $a_1 : 0.2$
$a_2 : 0.2$
$a_3 : 0.4$
$a_4 : 0.2$

⇒ Start initially with prob 0 to 1

- If we want to encode $a_1 a_2 a_3 a_3 a_4$




⇒ Once we have the final range $l_1$ to $r_1$, store the midpoint as the result. On decoding though we need to be careful and also send the length of message. Or we could add a special terminating character in our symbols list & stop when we see it.

- For decoding, using the value and keep checking in which range it lies, add that symbol & then expand that range & keep repeating till we get back the entire message.

(Note: We can also try to store $l_1$ or $r_1$ exactly so that when we see that for a certain range $l_x$ or $r_x$ is the same as our encoded value we stop.)

---

## *4) Dictionary Coding: (LZW Coding) (Lossless compression)

* • Dictionary need not be stored, unlike previous methods. (Example in slides and OSAA notes)

⇒ Great compression ratio especially when used on large documents etc.

(Unix compress uses LZW coding)

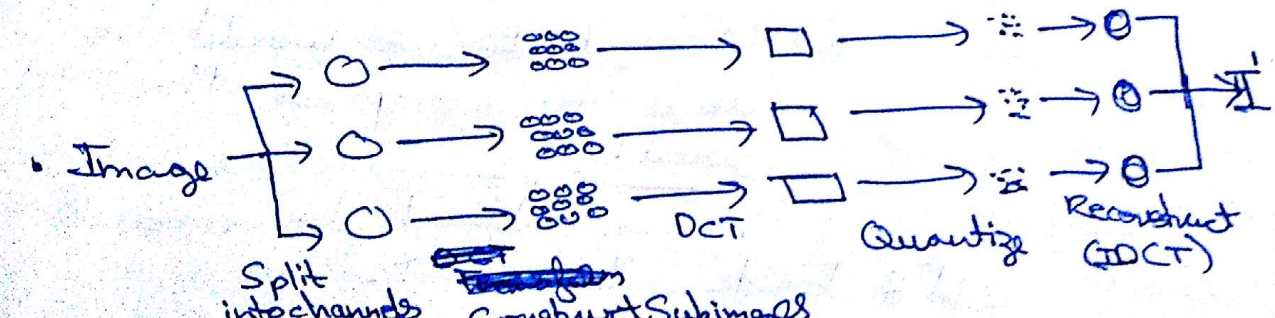
• We would need 1 or more additional bits to store the groups of characters we add to our dictionary.

Note: While decoding, if multilength strings need to be added, we only add entire first string + first character of second string. (Slides for example)

Note: If code is not present while decoding → we assume the last character repeats? ex: xxxxxx.

---

## *Lossy Compression JPEG: (OSAA Notes)



a) Why partition into 8x8 blocks?

↳ Since the color variation becomes minimal in this manner within each block

b) We can choose DCT or any other basis to perform the transformation.

* ↳ Subtract mean from image patch.
  ↳ We apply this transform to each 8x8 block
  ↳ We use DCT since trying different transforms, removing bottom 50% and then reconstructing, the least RMS etc errors occurred for DCT at window size = 16.
  ↳ 8x8 windows are used since in general it was used earlier so we just stick to it.

c) We get an 8x8 matrix after the transform and then divide that matrix with the standard 8x8 quantization matrix. (Slides)

↳ This would end up zeroing out a lot of the elements at the bottom right.

d) Perform a zig-zag scan starting from top left, to get a list of numbers.

↳ This is useful since we end up with lots of zeros at the end. (Good for compression)
↳ Using RLE we would be able to compress this very well.

---

Note: Before DCT we subtract the mean & add it back after IDCT.

e) After building back from RLE we get back the 8x8 matrix. We then dequantize by multiplying with standard 8x8 quantization matrix. We then perform IDCT and add back the mean.

---

Note: For color images we use Y, Cb, Cr channels and use low quantization on Y but large quantization on Cb, Cr (perception differences). These are essentially 3 8x8 matrices for luminance & chrominance.

⇒ These standard matrices were derived based on visual experiments.

---

# *Videos

- FPS
- Aspect Ratio
- Chroma subsampling → Bits per pixel.
- Compression algorithm (mp4, mpeg etc)
- Compression container

* Interlaced vs Progressive
  - Interlaced: one frame stores even rows whereas the other stores odd columns and so on. (Post processing is hard)
  - Progressive: Complete info stored in each frame

---

# *Block Matching for Compression:

* Given frame t and frame t-1, we divide frame t into blocks and for each block we see which pixels in t-1 correspond the most to this.

⇒ This would give each block a flow vector. Now using frame at t-1 & the flow we can reconstruct an approx
(This won't be exact since we only get closest matches from previous frame if it exists there (ie. translations) but deformations etc cannot be captured)

⇒ The solution is to store another image which is the difference b/w frame t and reconstructed frame t.
(Here most of the values are zero since changes in frame to frame are minimal therefore most of the matches are perfect)

— x —

## * Matching: (Blocks)

### 1) Exhaustive search:

- For each block in frame t, search in a local neighbourhood of frame t+1

⇒ Things like squared error, correlation etc.

### 2) Pyramid search: (Search overall first, then a part of the image and so on)

### 3) Logarithmic search: (Slides)

- If we have a convex function on the image, we can move according to the gradients.

- Convex gives global solution but generally it doesn't work.

⇒ This is fine though since we store differences anyway.

———x———

Note: Pixel sub sampling:

- While comparing patches, dont consider all pixels of a patch & maybe just use odd rows & odd columns etc.

Pixel projection:

- Sum up rows or columns & check for these matches etc. (Slides)

⇒ These give us computational benifits.

———x———

* Sub pixel motion estimation:

- Upsample frames and then compute flows etc so that single pixels in this correspond to 1/2 pixels in the original frames.

⇒ So we get flows for half pixels etc.

———x———

Hierarchical
(The most popularly used)

———x———

# Video Compression:

* • M-JPEG, use JPEG on each frame directly and just store them (An old approach).

⇒ Block Matching → MPEG uses 16x16 blocks. → Transform + Quantizing → Encoding → Compressed Video

* Differences

* (Pyramid flow + differences) → Hierarchial

(Similarly we decompress the video).

—— x ——

⇒ Nowadays, we have 3 types of frames I, B, P. I = Actual frame
B = Bilinear interpolated
P = Predicted using block matching etc.

(Slides for example)

I → B → B → P → B → B → P → I → ...

(We add I's in the middle to limit the error etc).

—— x ——

**

## 1) H.264 Compression: (Much better than MPEG)

a) • They optimized and used variable sized blocks.

⇒ Start with 16x16 and if variance is too high break it into 8x8 blocks or 4x4 blocks.

b) Quarter pixel accuracy, so they upsampled twice from original size. More compression achieved.

c) Multiple frames used for prediction.
d) Integer transform instead of DCT.

---

# *Content Aware Image Resizing:

- Image resizing → Cropping, Scaling., Letterboxing
  - Cropping → (Pan and Scan in videos)
  - Scaling. → Content Aware Image Resizing.
  - Letterboxing → (Add black strips on ends) (To get required ratio) ← (wastes screenspace)

## 1) Seam Carving: (Examples in slides)

* Compute characteristics of image, ex: Gradient.
⇒ Then select a seam which has minimum edges on it and remove that seam.
(we could also use other characteristic features etc instead of the gradients).

---

Note: We could just remove rows or columns by themselves but it creates artefacts, so we choose seams.

---

- To identify a seam, we need to find a connected path from top to bottom.
For each pixel go to all 3 below neighbours try bruteforce all paths & choose minimum. This can be optimized by a DP.

* ⇒ $DP[i,j] = \min(DP[i-1,j-1], DP[i-1,j], DP[i-1,j+1]) + edge(i,j)$

(This is if we want seams from top to bottom)
(we could also go from left to right for row seams).

- So in the last row, we check which pixel has minimum value of DP and remove that pixel & move everything to the right of it 1 pixel to the left. We now go to the parent which gave us the minimum DP and repeat this process. At the end we remove the last column.

(We end up removing 1 pixel in each row, so this is removing a column seam from the image)

---

Note: Instead of just edge image, we can also include saliency based on what we think is important etc.

Note: Image can also be expanded, choose minimal seams and then add those seams to the image and make pixel values average of left, right (for column seams).

---

Note: We can also remove certain things from the image. User marks that thing & in the edge map we give it a very large -ve value and we then remove seams until that thing is gone & then we expand back to the original size.

(Similarly we can give the weights to things we don't want to lose).

---

Note: This can be extended to videos, but we need to make sure temporal changes are taken care of.

---

Note: Shift map editing, Patch match, content aware warping

(Papers for content aware resizing)

## 2) Texture Synthesis: (Application of seam carving)

* We have a small image patch, we need to generate a larger patch.

⇒ Take 2 patches and put them next to each other, horizontally with some overlap such that SSD over overlap is minimum.

(Patches are small parts of the image which capture most of the information we need).

* Now in that overlap region, compute importance matrix as SSD (Similar to edge image in seam carving). We then choose a seam in this overlap and then everything to the left we use first patch and everything to right of seam we use second path.

⇒ We can then repeat this process in the vertical direction as well.

(examples in slides).

---

(Note: More examples in slides, some more ** extensions etc. Generate an image based on a given texture and content image)

---

# *Chamfer Matching:

- Given a template sketch & a query image.
  ↳ Find where the template occurs.

## 1) Edge Image & SSD:

* - Compute edge image of query image.
  - Compute SSD by sliding template over query image edges.

⇒ This won't work that well.

## *2) Distance Transform:

* - For each pixel in the edge image of query image, value = distance from nearest edge pixel.
  (This is similar to a gaussian blur on query image. We do this after getting edges of query image)
  - Once we have the distance transform of edge image of query image, we perform standard SSD kind of thing ~~where the value of pixel is large if its closest to edge exist / distance image etc.~~ If the value is small then its a good match. (Or we can consider the inverse etc)

⇒ We can ensure scale issues are taken * care of by using pyramids. Try some rotations etc, for invariance

- Distance Transform Algorithm:

* * ↳ Initialize edges to 0 and all other pixels are ∞ in the dp.
  ↳ We then move rights & down first compute

$$dp[i,j] = \min(dp[i,j-1]+1, dp[i-1,j], dp[i,j])$$

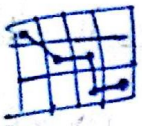
↳ we then move left & top computing
$dp[i,j] = \min(dp[i,j+1], dp[i+1,j], dp[i,j])$

- We would then have our distance transform stored in dp image.

(This works since all paths that go right & down are captured in the first iteration & all paths that go left & top are captured in the second backwards iteration)

———x———

Note: If we want to use checkerboard distance (allow diagonals) then we do $dp[i,j] = \min(dp[i, j-1], dp[i-1,j], dp[i-1,j-1], dp[i,j])$ (motion)



and similar for the second backwards pass.

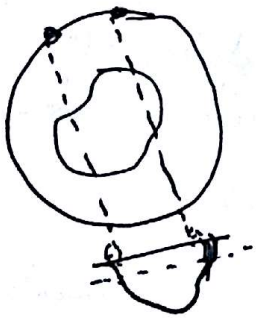
———x———

Note: How can we get distance transform with euclidean distance?

———x———

# Image Reconstruction
(From projections)

## *CT Scan: (Computed Tomography)



- Pass beams rotating around the object (voxels in medical)
- Calculate the output as sum of densities along path.

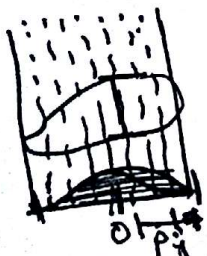
⟹ Projection reconstruction: (examples in slides)

- Each of these projected densities move them along the path of the rays adding them to all the pixels/voxels along their path.
- Repeat this for different angles and you will start to get back the original image/object.

———— x ————

## *Radon Transform:

- The density integral along each line of projection is calculated using this transform.
- So its essentially adding up all the pixels along that line for each line.




} This is the projection

⟹ Suppose we consider rays normal at an angle $\theta_k$ from the x-axis,.

* $g(\rho_j, \theta_k) = \iint f(x,y)\,\delta(x\cos\theta_k + \sin\theta_k - \rho_j)\,dx\,dy$

$\rho_j$: Distance from the center on the projection.

$\theta_k$: Angle of normal of projection

---

Note: Sinograms: For each angle we compute the radon transform and place them one on top of another to get something like a spectrogram.

* (examples in slides)

(2D sinogram for 2D images, and 3D for 3D)

---

$$\Rightarrow f(x,y) = \int_{\theta=0}^{\pi} f_\theta(x,y) = \int_{\theta=0}^{\pi} g(\rho,\theta_k)$$

$$* \quad f(x,y) = \int_{\theta=0}^{\pi} g(x\cos\theta_k + y\sin\theta_k, \theta_k)$$

↑ Reconstruction

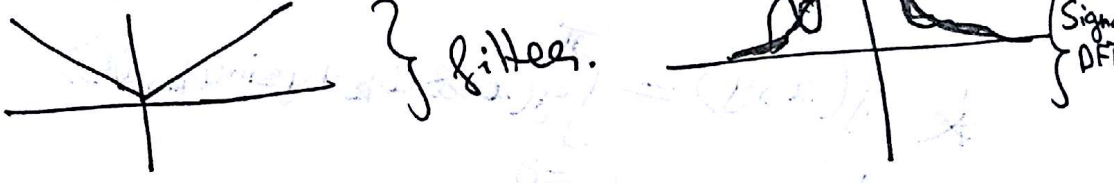
---

## *Fourier Slice Theorem: (example in slides)

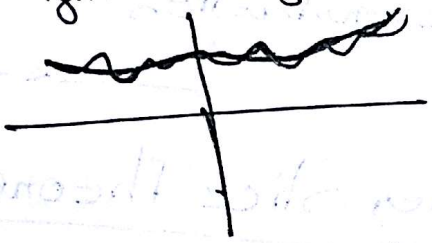
* • At a normal angle of $\theta$ for each projection we compute the radon transform.
  • Now a 1D fourier transform of this is the same as the line in the 2D fourier transform of the entire original image which is at an angle $\theta$ from x-axis through 0.

⇒ So now if we repeat this process for all angles we would be able to create our 2D fourier transform of original image, but the issue here is that the density of datapoints at lower frequencies (near zero) is much higher than the higher frequencies (So on direct projection reconstruction we get blurred images)

⇒ The way we fix this is by using filtered backprojection. We apply a filter which enlarges high frequencies and minimizes low frequencies




- Final_signal DFT = filter X Signal DFT



⇒ We now perform back projection using this final_signal by computing IDFT of final_signal DFT and this would give us much sharper images.