Efficient filters + linear algebra review

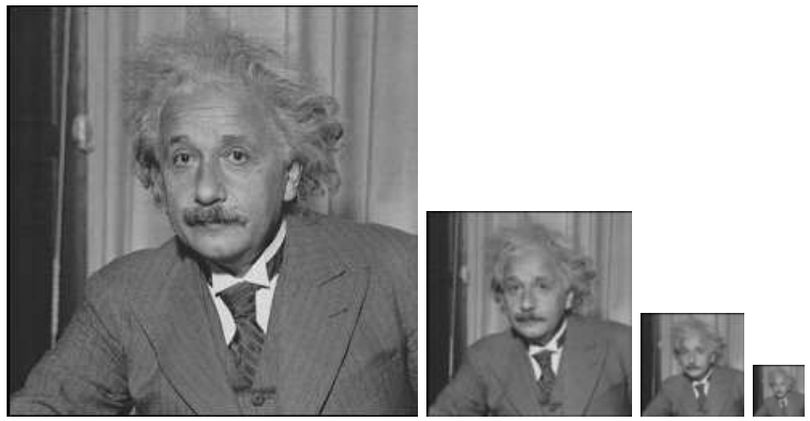
Logistics

Date	Topic	References	Misc
1/14	Introduction		Linear Algebra Review
1/16	Filters		Special Recitation on HW0; NSH 1305 at 5pm
1/21	Edges		HW0 - Python Intro (not graded)
1/23	Texture		
1/28	Linear algebra review		
1/30	Warping		HW1 - HOG Image Classification
2/4	Alignment		
2/6	Frequency	Guest Lecture (tentative)	
2/11	Correspondence		
2/13	Descriptors		HW2 - LK Tracking

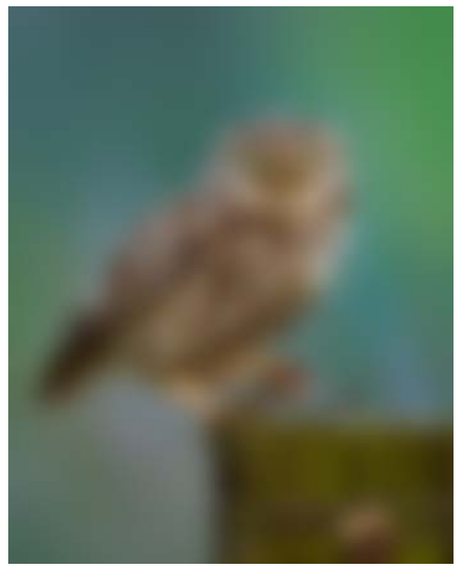
Outline

- Convolution
  - Linear Shift Invariance (LSI)
  - Convolution Properties (commutative, associative, distributive)
  - Normalize Cross-Correlation
- Edges
  - Canny edges (hysterisis)
  - derivative-of-gaussians (DoG)
  - laplacian-of-Gaussians (LoG)
  - filter banks
- **Efficiency**
  - pyramids
  - linear approximiations (SVD, separability)
  - steerability

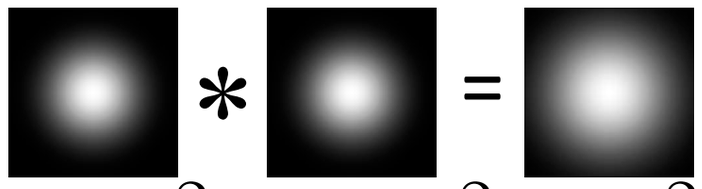
Pyramids



- Big filters (e.g., Gaussians) tend to be smooth, so the output is redundant



- Exploit property that Gaussian*Gaussian = Bigger Gaussian



$$\sigma_{a*b}^2 = \sigma_a^2 + \sigma_b^2$$

Proof: https://en.wikipedia.org/wiki/Sum_of_normally_distributed_random_variables

The Laplacian Pyramid as a Compact Image Code

PETER J. BURT, MEMBER, IEEE, AND EDWARD H. ADELSON

Abstract—**We describe a technique for image encoding in which local operators of many scales but identical shape serve as the basis functions. The representation differs from established techniques in that the code elements are localized in spatial frequency as well as in space.**

Pixel-to-pixel correlations are first removed by subtracting a low-pass filtered copy of the image from the image itself. The result is a net data compression since the difference, or error, image has low variance and entropy, and the low-pass filtered image may represented at reduced sample density. Further data compression is achieved by quantizing the difference image. These steps are then repeated to compress the low-pass image. Iteration of the process at appropriately expanded scales generates a pyramid data structure.

The encoding process is equivalent to sampling the image with Laplacian operators of many scales. Thus, the code tends to enhance salient image features. A further advantage of the present code is that it is well suited for many image analysis tasks as well as for image compression. Fast algorithms are described for coding and decoding.

does not permit simple sequential coding. Noncausal approaches to image coding typically involve image transforms, or the solution to large sets of simultaneous equations. Rather than encoding pixels sequentially, such techniques encode them all at once, or by blocks.

Both predictive and transform techniques have advantages. The former is relatively simple to implement and is readily adapted to local image characteristics. The latter generally provides greater data compression, but at the expense of considerably greater computation.

Here we shall describe a new technique for removing image correlation which combines features of predictive and transform methods. The technique is noncausal, yet computations are relatively simple and local.

The predicted value for each pixel is computed as a local

Peter J. Burt (M'80) received the B.A degeree i physics from Harvard University, Cambridge, MA in 1968, and the M.S. and Ph.D. degrees i computer science from the University of Massa chusetts, Amherst, in 1974 and 1976, respec-tively

From 1968 to 1972 he conducted research i sonar, particularly in acoustic imaging devices a the USN Underwater Sound Laboratory, Nev London, CT and in London, England. As Postdoctoral Fellow, he has studied both natural vision and computer imag understanding at New York University, New York, NY (1976-1978), Be Laboratories (1978-1979), and the University of Maryland, College Par (1979-1989). He has been a member of the faculty at Rensselae Polytechnic Institute, Troy, NY, since 1980.

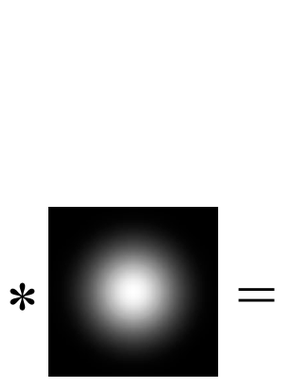
Edward H. Adelson received B.A. degree in physics and philosophy from Yale University, New Haven, CT, in 1974, and the Ph.D degree in experimental psychology from the University of Michigan, Ann Arbor, in 1979.

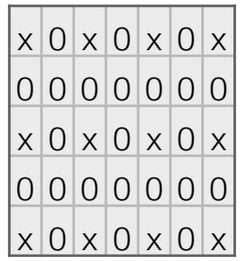
From 1979 to 1981 he was Postdoctoral Fellow at New York University, New York, NY. Since 1981, he has been at RCA David Sarnoff Research Center, Princeton, NJ, as a member of the Technical Staff in the Image Quality and Human Perception Research Group. His research interests center on visual processes in both human and machine visual systems, and include psychophysics, image processing, and artificial intelligence.

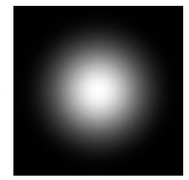
Dr. Adelson is a member of the Optical Society of America, the Association for Research in Vision and Opthalmology, and Phi Beta Kappa.

Key idea: rather than convolving with big gaussian

Convolve with small gaussian, subsample, convolve with small gaussian, subample,….



Can we make even more efficient (since we know we will be subsampling every-other-row-and-column the output)?

yes - perform *strided* convolution that shifts filter by 2 pixels instead of 1

Burt & Adelson, 83

Implementation in 1D

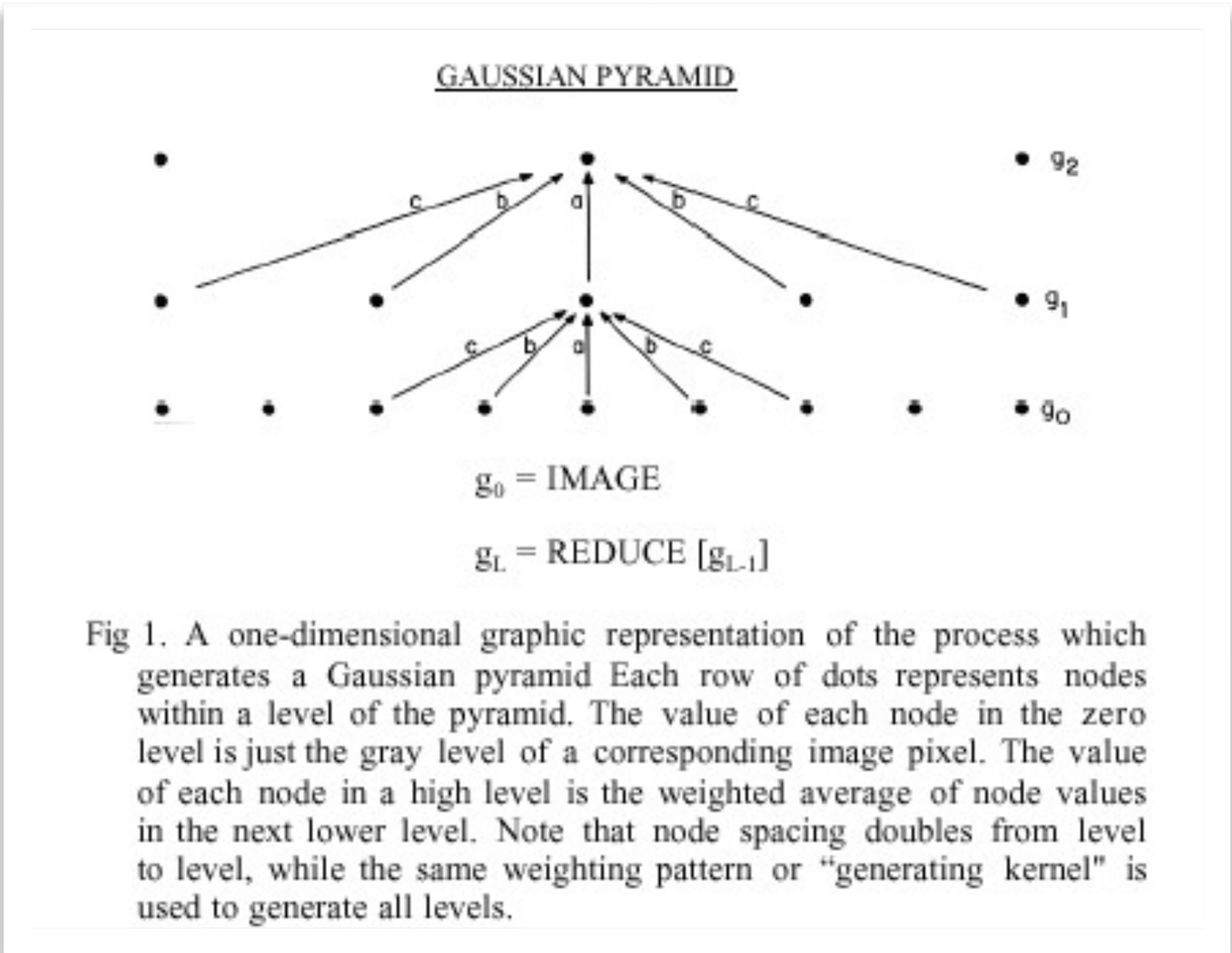
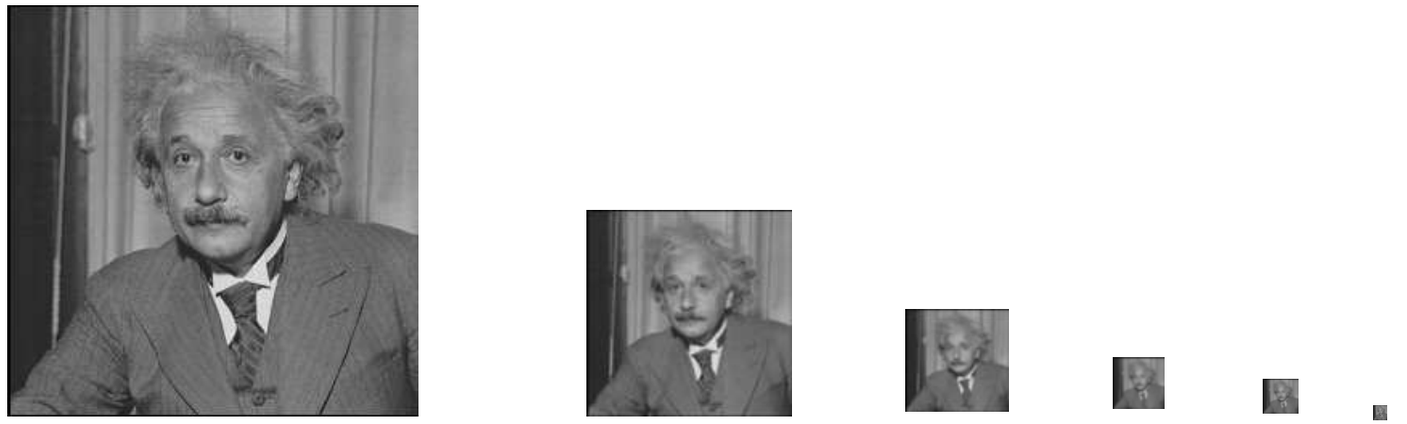


Fig 1. A one-dimensional graphic representation of the process which generates a Gaussian pyramid Each row of dots represents nodes within a level of the pyramid. The value of each node in the zero level is just the gray level of a corresponding image pixel. The value of each node in a high level is the weighted average of node values in the next lower level. Note that node spacing doubles from level to level, while the same weighting pattern or "generating kernel" is used to generate all levels.

$$H[i] = \frac{1}{16}\begin{bmatrix}1 & 4 & 6 & 4 & 1\end{bmatrix}$$

5-tap 1D Gaussian filter



First 6 levels of a Gaussian Pyramid. Original image is 257x257; level 5 is 9x9

$$REDUCE(F)[i] = \sum_{u=-2}^{2} H[u]F[2i+u]$$

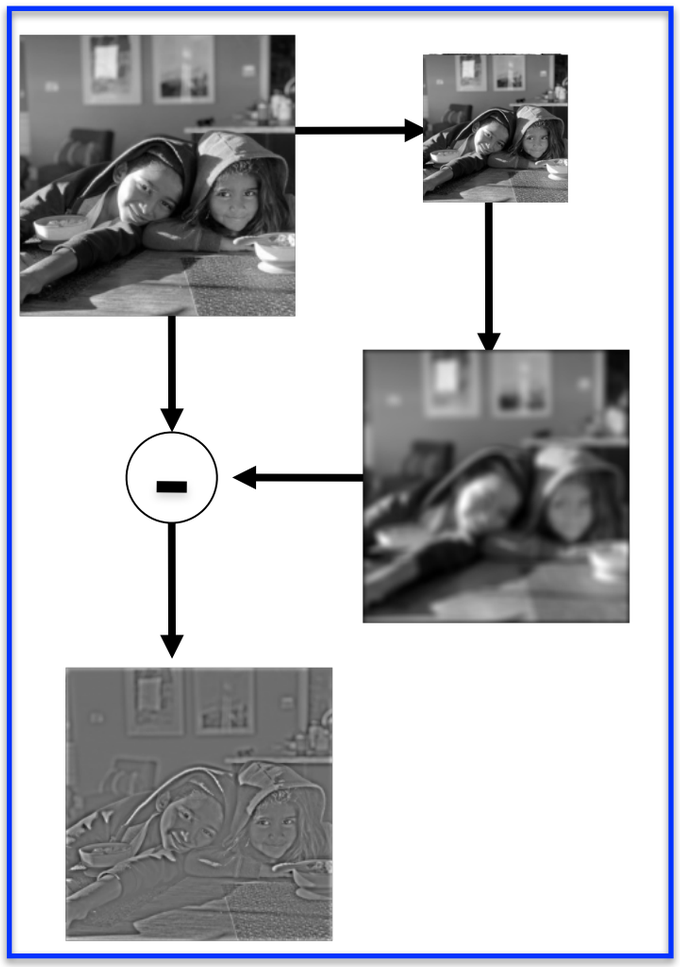
$$H[i] = \frac{1}{16}\begin{bmatrix}1 & 4 & 6 & 4 & 1\end{bmatrix}$$

```
def reduce(image):
# Reduce size of image by 2X by blurring and subsampling
  k = np.array([1,4,6,4,1])/16
  kernal = np.outer(k,k)
  res = np.zeros((image.shape[0]//2,image.shape[1]//2,image.shape[2]))
  for i in range(image.shape[2]):
    res[:,:,i] = signal.convolve2d(image[:,:,i], kernal, mode='same')[::2,::2]
  return res
```

Does this perform strided convolution? No, so REDUCE can be made more efficient

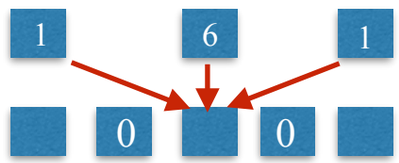
Laplacian pyramid

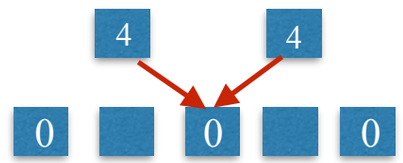
Store *difference* between upsampled image and original image



Upsample by inserting zeros between pixels and apply renormalized Gaussian filter:

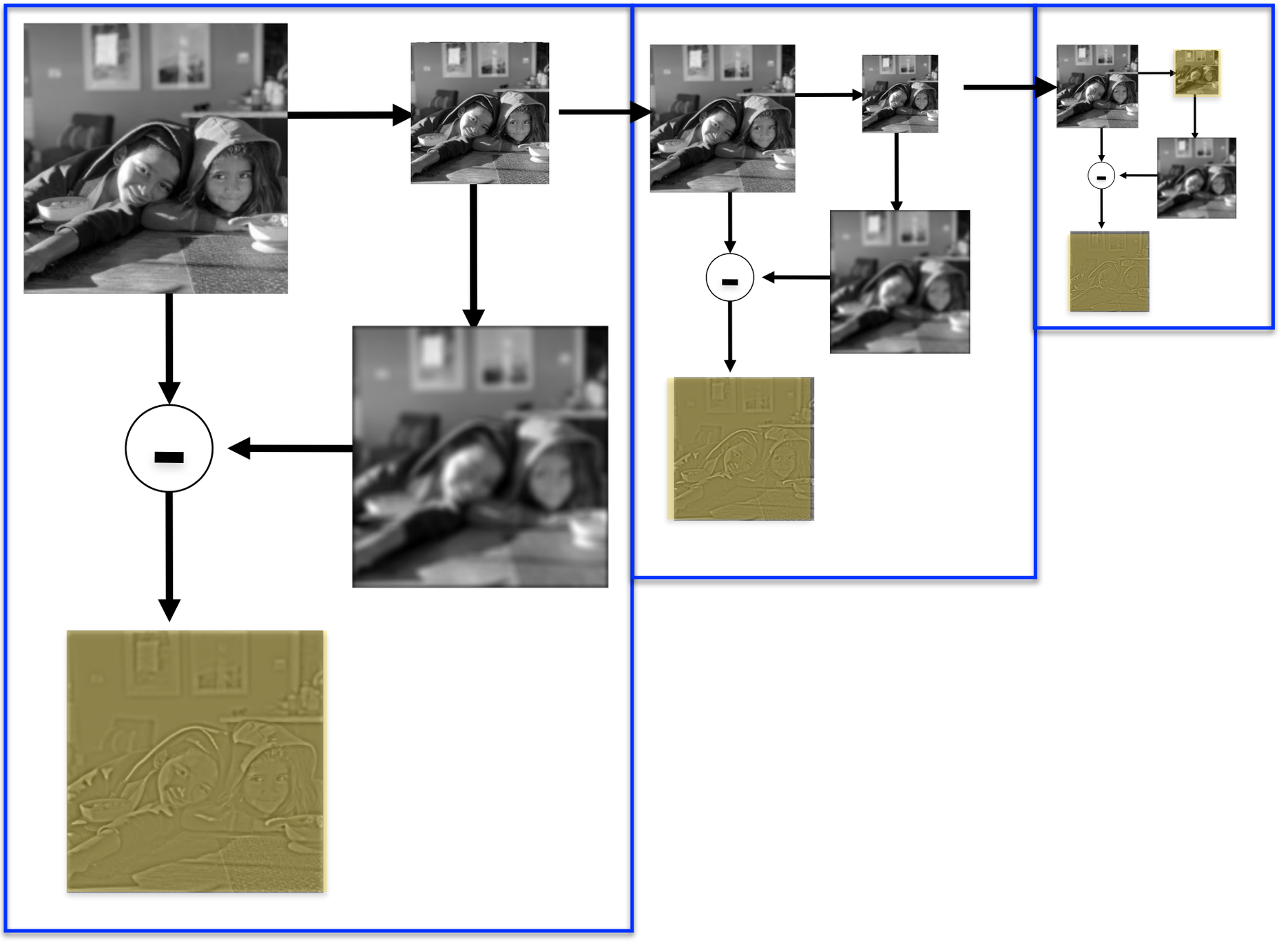
$$EXPAND(F)[i] = 2 \sum_{u=-2}^{2} H[u]F[(i+u)/2]$$





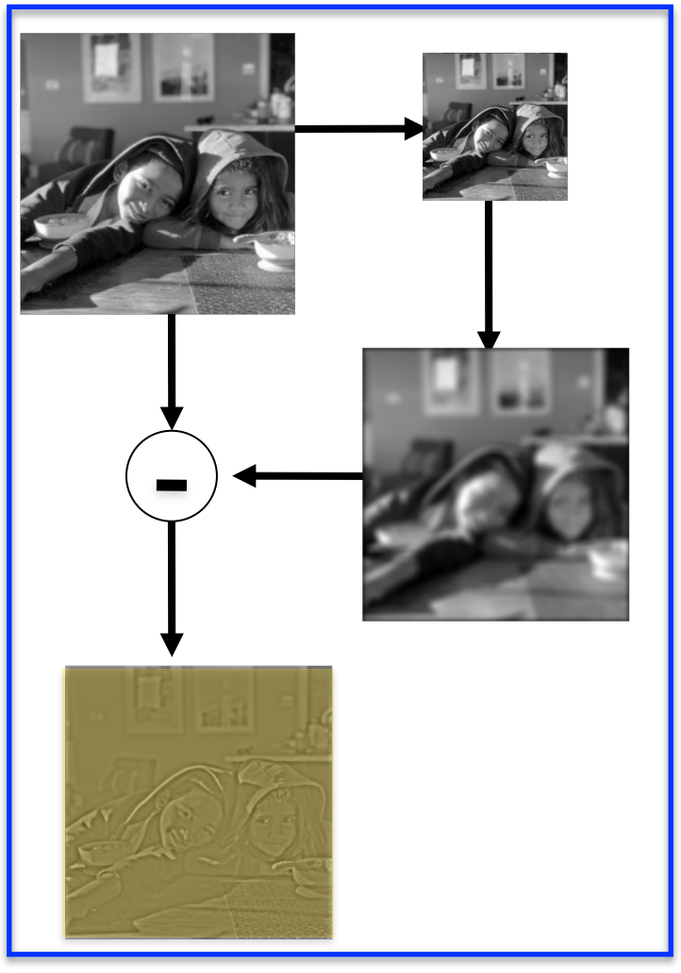

```
def expand(image):
# Expand image size by 2X by zero-interlacing and weighted-averaging
  k = np.array([1,4,6,4,1])/16
  kernal = np.outer(k,k)
  res = np.zeros((2*image.shape[0],2*image.shape[1],image.shape[2]))
  res[::2,::2,:] = image
  for i in range(image.shape[2]):
    res[:,:,i] = signal.convolve2d(res[:,:,i],4*kernal, mode='same')
  return res
```

Laplacian pyramid



Only need to store tiny-res image + sparse (mostly zero) delta-images

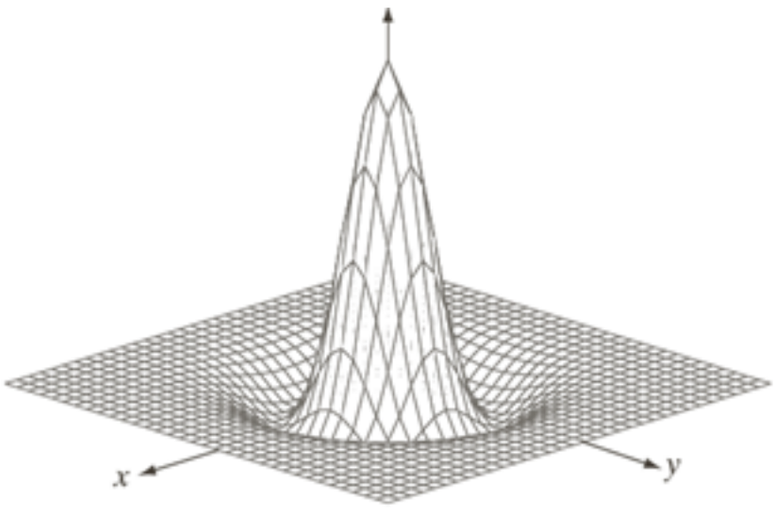
Laplacian pyramid



Can we directly produce the difference image with a linear filter?

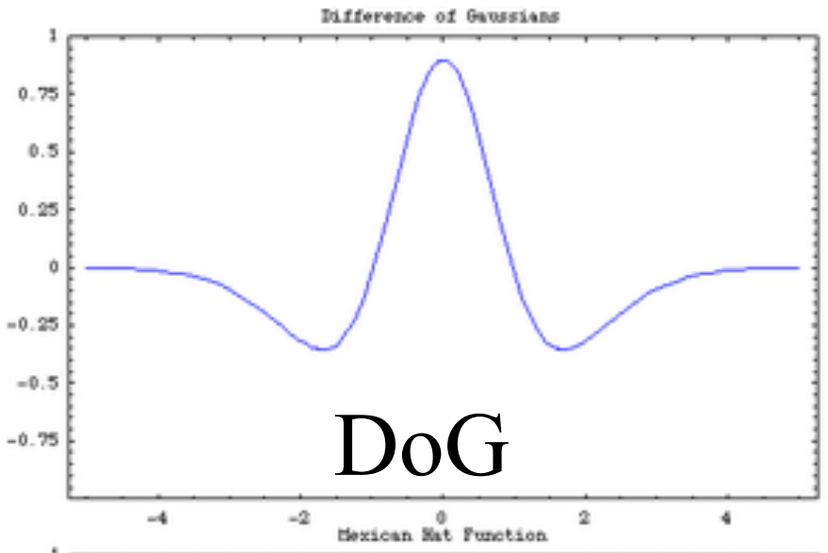
Difference of Gaussian (DoG) filter

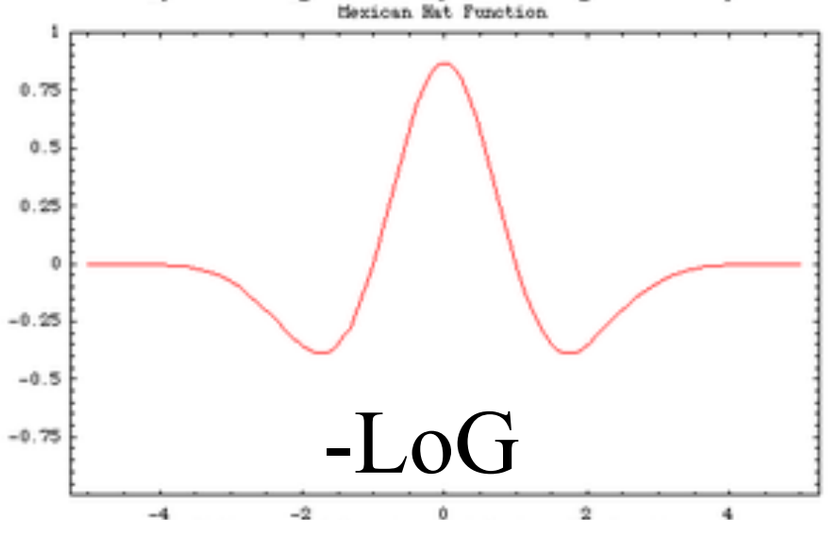
Looks very similar to a laplacian of Gaussian filter, but a different derivation!



$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

(derived as sum of second derivative in x and y)

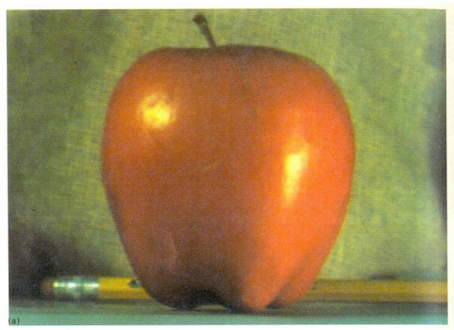




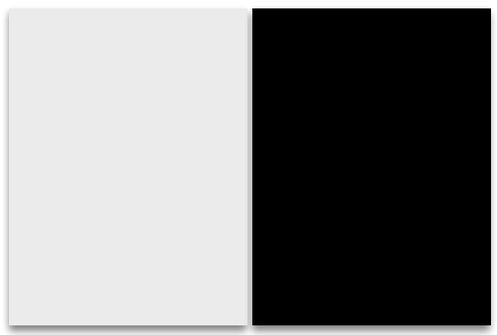
Multiresolution blending

demo_blending.ipynb

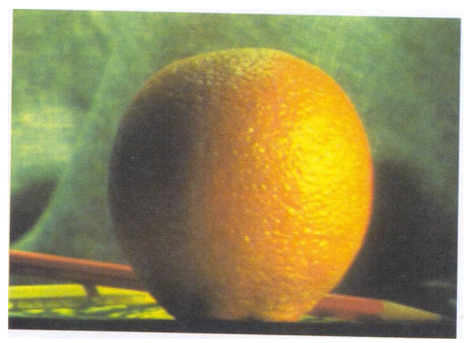
Left image

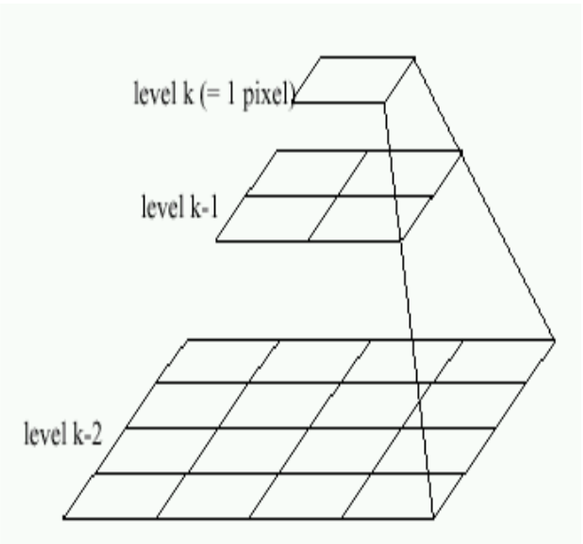


Blend mask

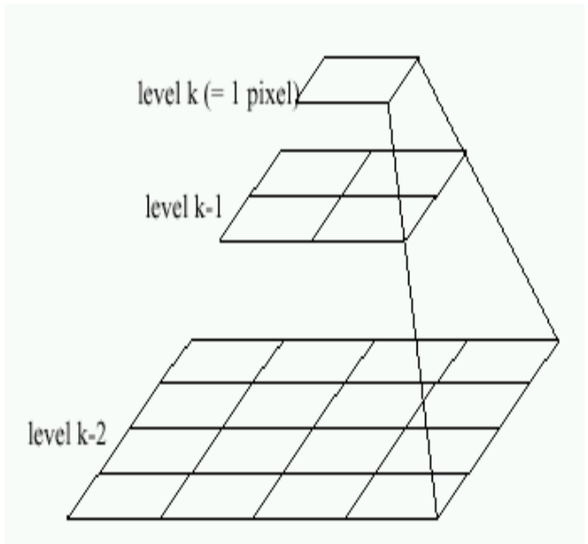


Right image

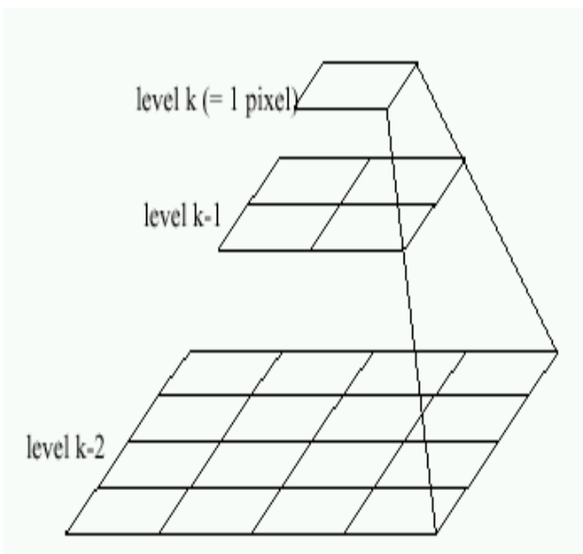




Laplacian pyramid



Gaussian pyramid

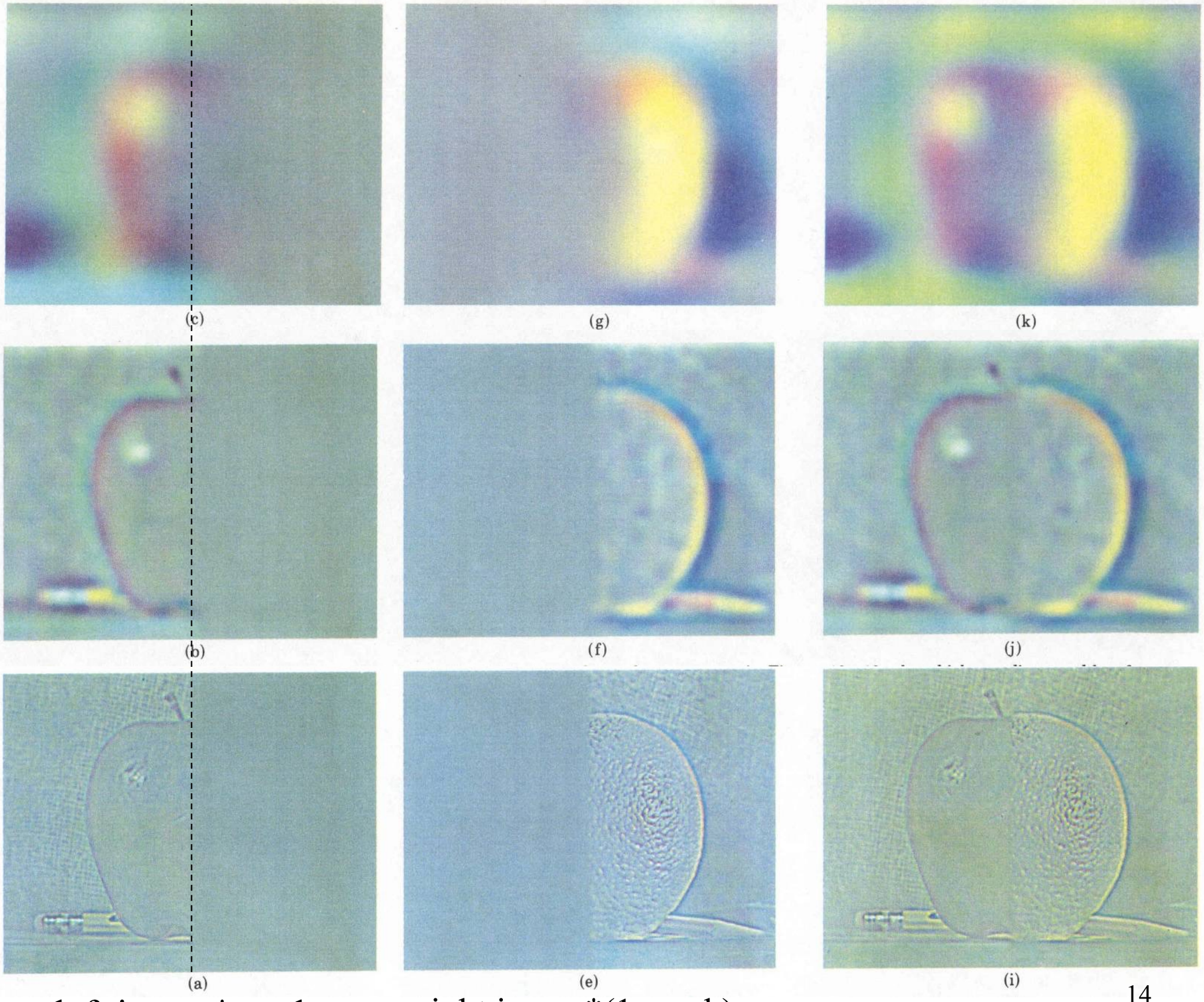


Laplacian pyramid

laplacian level 4

laplacian level 2

laplacian level 0

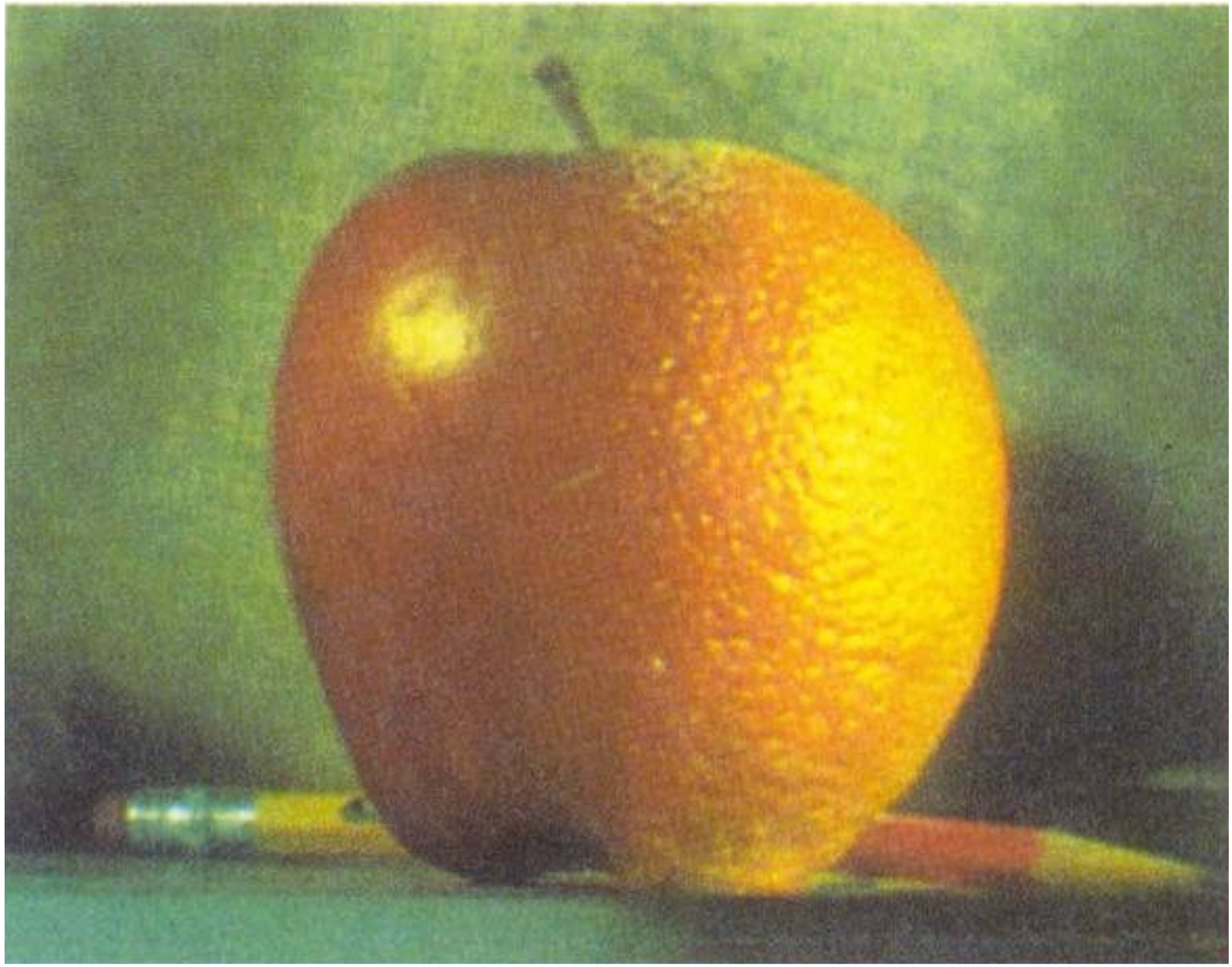


left image*mask

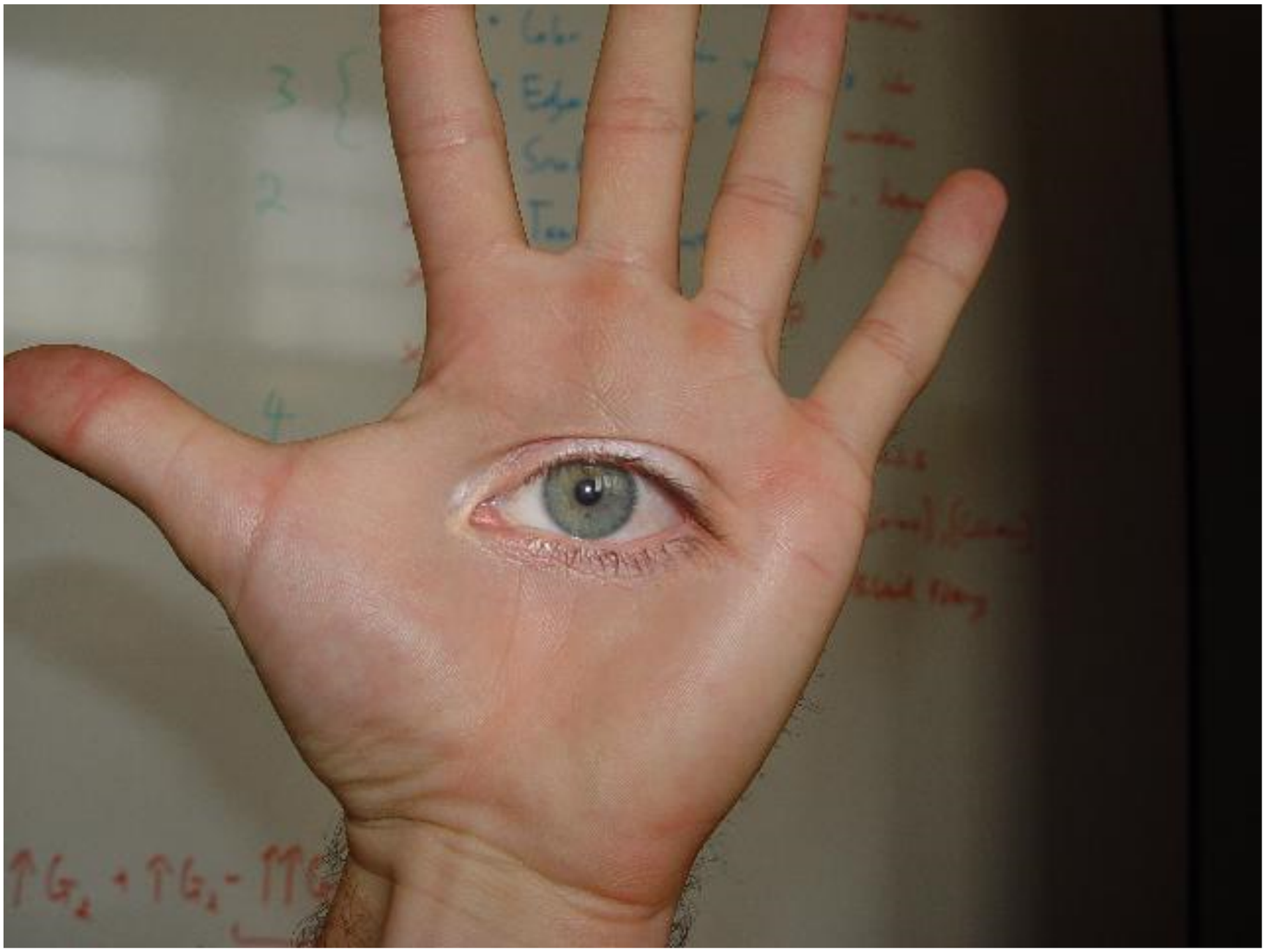
right image*(1-mask)

sum

Pyramid Blending



Fun example!



© david dmartin (Boston College)

Outline

- Convolution
  - Linear Shift Invariance (LSI)
  - Convolution Properties (commutative, associative, distributive)
  - Normalize Cross-Correlation
- Edges
  - Canny edges (hysterisis)
  - derivative-of-gaussians (DoG)
  - laplacian-of-Gaussians (LoG)
  - filter banks
- Efficiency
  - pyramids
  - **integral images** / separability
  - linear approximiations (SVD, separability)

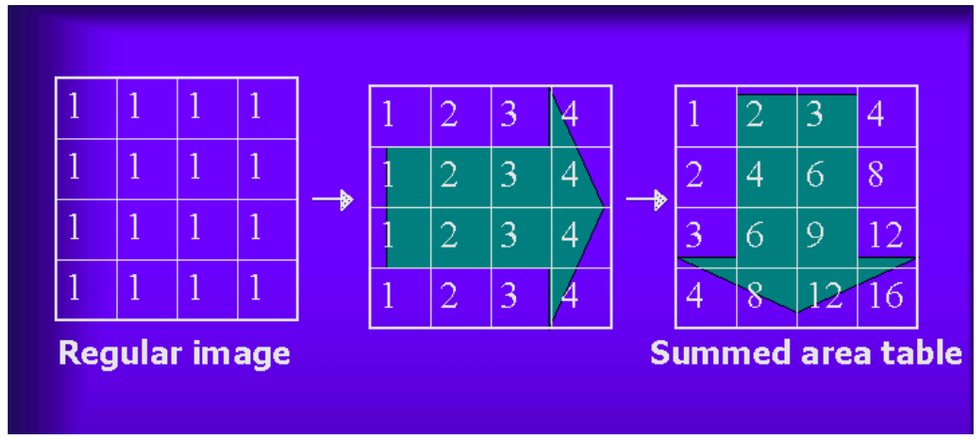
Cute trick for efficient *box* filters

Consider a filter of all “1”’s (which sums up values in a neighborhood)

Turns out there is a way to implement filter that is *independant* of filter size

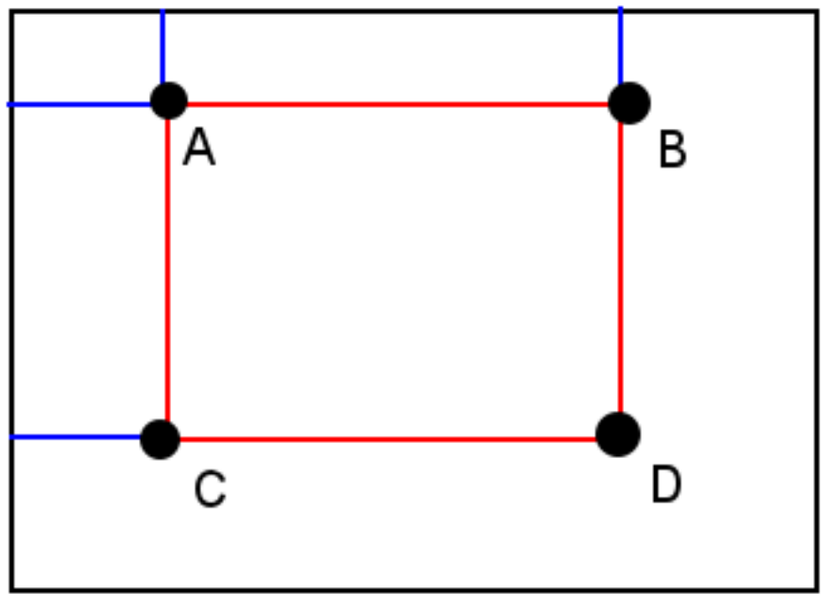
Box filtering with integral images (or summed area tables)

http://en.wikipedia.org/wiki/Summed_area_table



$$I(x,y) = \sum_{\substack{x' \le x \\ y' \le y}} i(x',y')$$

Given image i(x,y) and its integral image I(x,y),
how can we efficiently compute the sum of all pixels from i(x,y) within red rectangle?



Sum = D - B - C + A

Compute filter response for any size window with 4 table lookups (in integral image)!

Reduces $O(N^2M^2)$ to $O(N^2)$

Efficient detection

P. Viola and M. Jones. *Rapid object detection using a boosted cascade of simple features*. CVPR 2001.

Outline

- Convolution
  - Linear Shift Invariance (LSI)
  - Convolution Properties (commutative, associative, distributive)
  - Normalize Cross-Correlation
- Edges
  - Canny edges (hysterisis)
  - derivative-of-gaussians (DoG)
  - laplacian-of-Gaussians (LoG)
  - filter banks
- Efficiency
  - pyramids
  - integral images
  - ~~**steerability**~~ separability
  - linear approximiations (SVD, separability)

Separability

Image of size N^2
Filter of size M^2

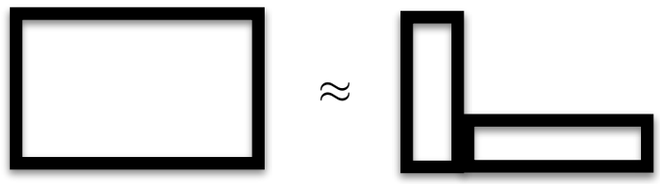
Complexity of filtering? (let's stick with correlation for now)

$O(N^2M^2)$

$$H[u,v] = H_x[u]H_y[v]$$

$$G[i,j] = \sum_u \sum_v H[u,v]F[i+u, j+v]$$

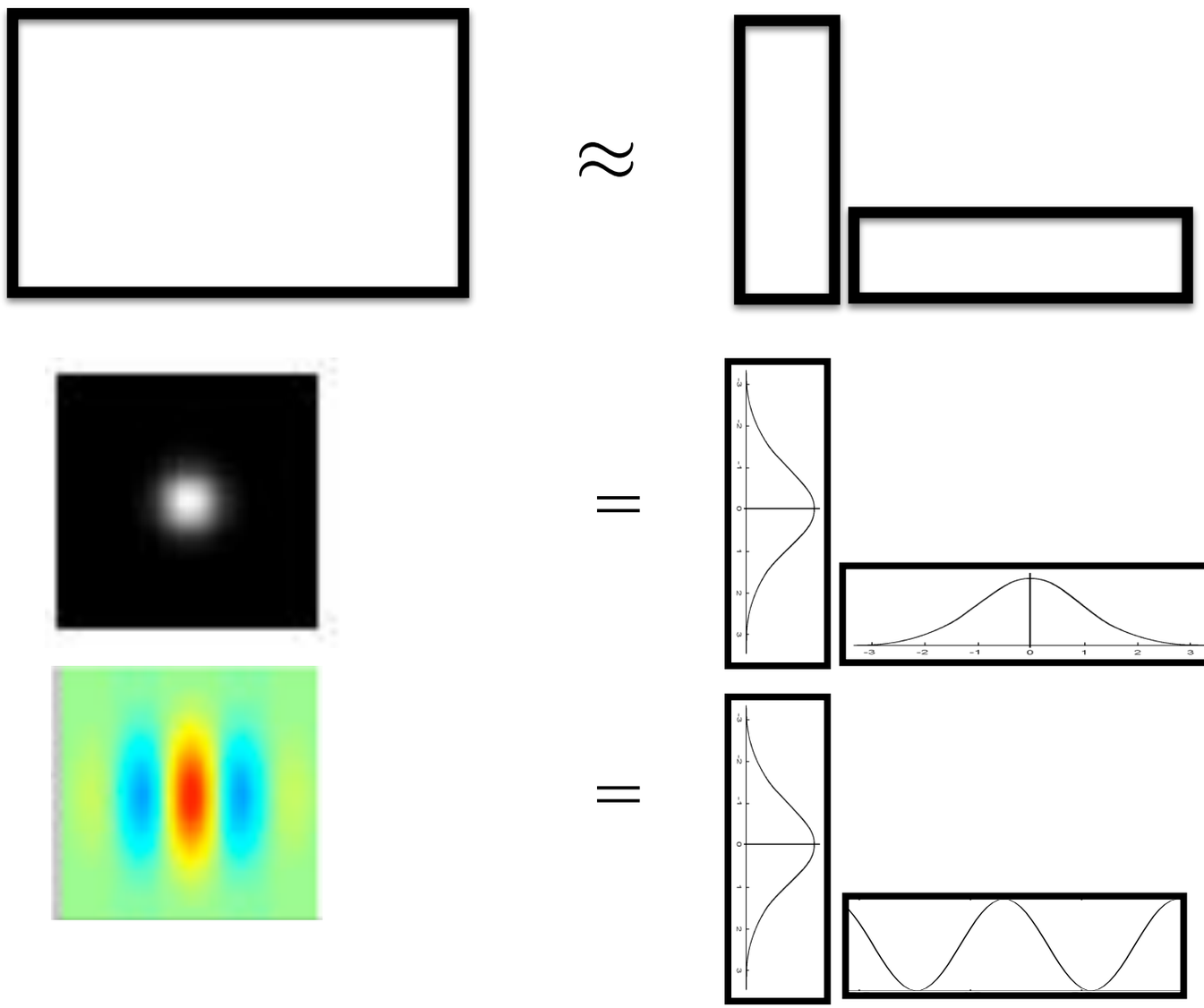
$$=$$



Separability

Given a filter, how can we come up with a good separable approximation?

$$H[u, v] \approx H_x[u] H_y[v]$$



We'll make use of *low-rank decompositions* from linear algebra

Linear algebra digression

https://www.cs.cmu.edu/~zkolter/course/linalg/linalg_notes.pdf

Linear Algebra Review and Reference

Zico Kolter (updated by Chuong Do)

October 7, 2008

Contents

My own writeup (svd.pdf in lec gdrive folder):

https://drive.google.com/file/d/1c-kjMqIQEKJCHyfMgLzdxRJNReOu6ssk/view?usp=drive_link

SVD

Deva Ramanan

January 31, 2024

Let us represent a linear transformation as follows:

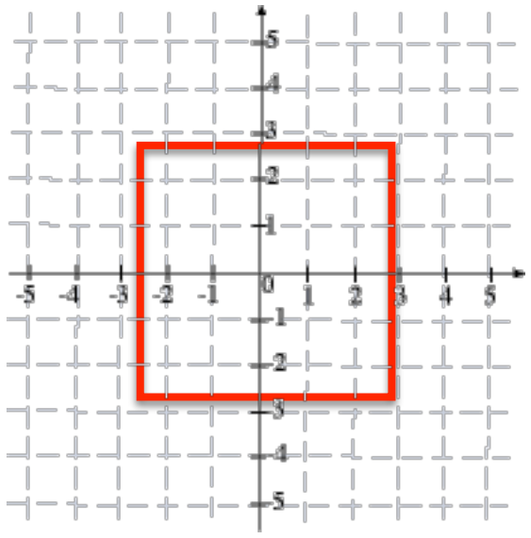
$$\mathbf{y} = A\mathbf{x}, \quad A \in R^{n \times m}, \mathbf{x} \in R^m, \mathbf{y} \in R^n \tag{1}$$

where A is a matrix with n rows and m columns. This document uses the singular value decomposition (SVD) to decompose A into a series of geometric transformations, focusing intuition rather than a precise formulation. For simplicity, let $n = 3$ and $m = 2$, such that A transforms points in 2D to 3D:

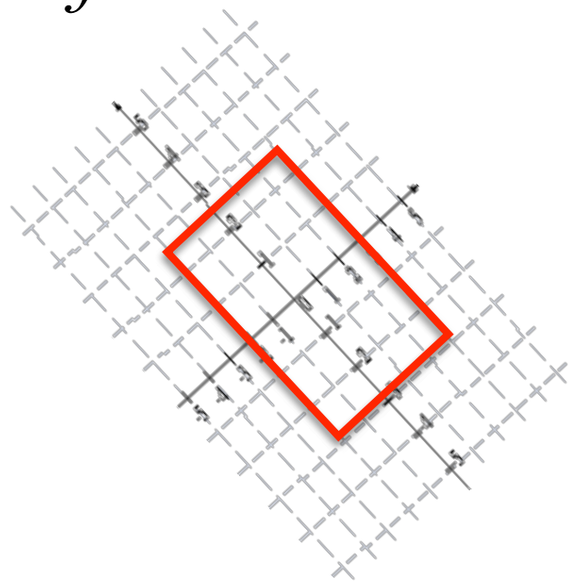
$$\mathbf{y} = A\mathbf{x}, \quad A \in R^{3 \times 2}, \mathbf{x} \in R^2, \mathbf{y} \in R^3 \tag{2}$$

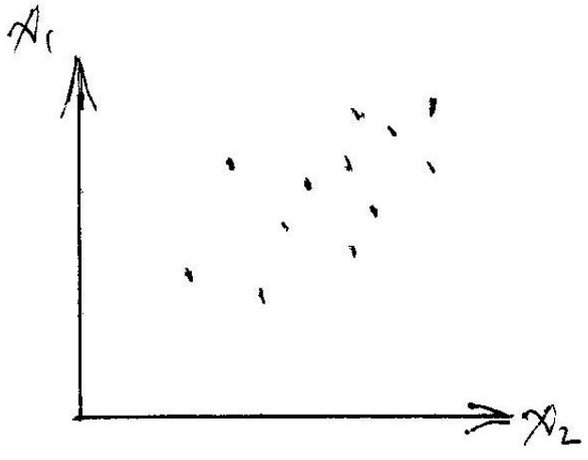
Linear algebra digression

Any matrix can be thought of as a *transformation*

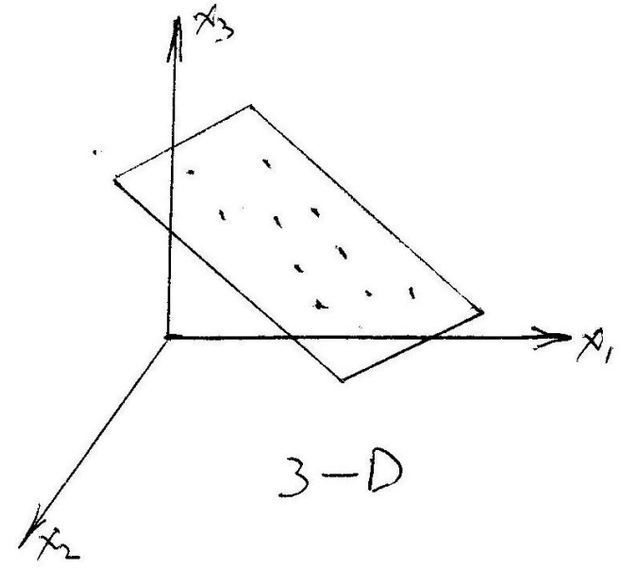


$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

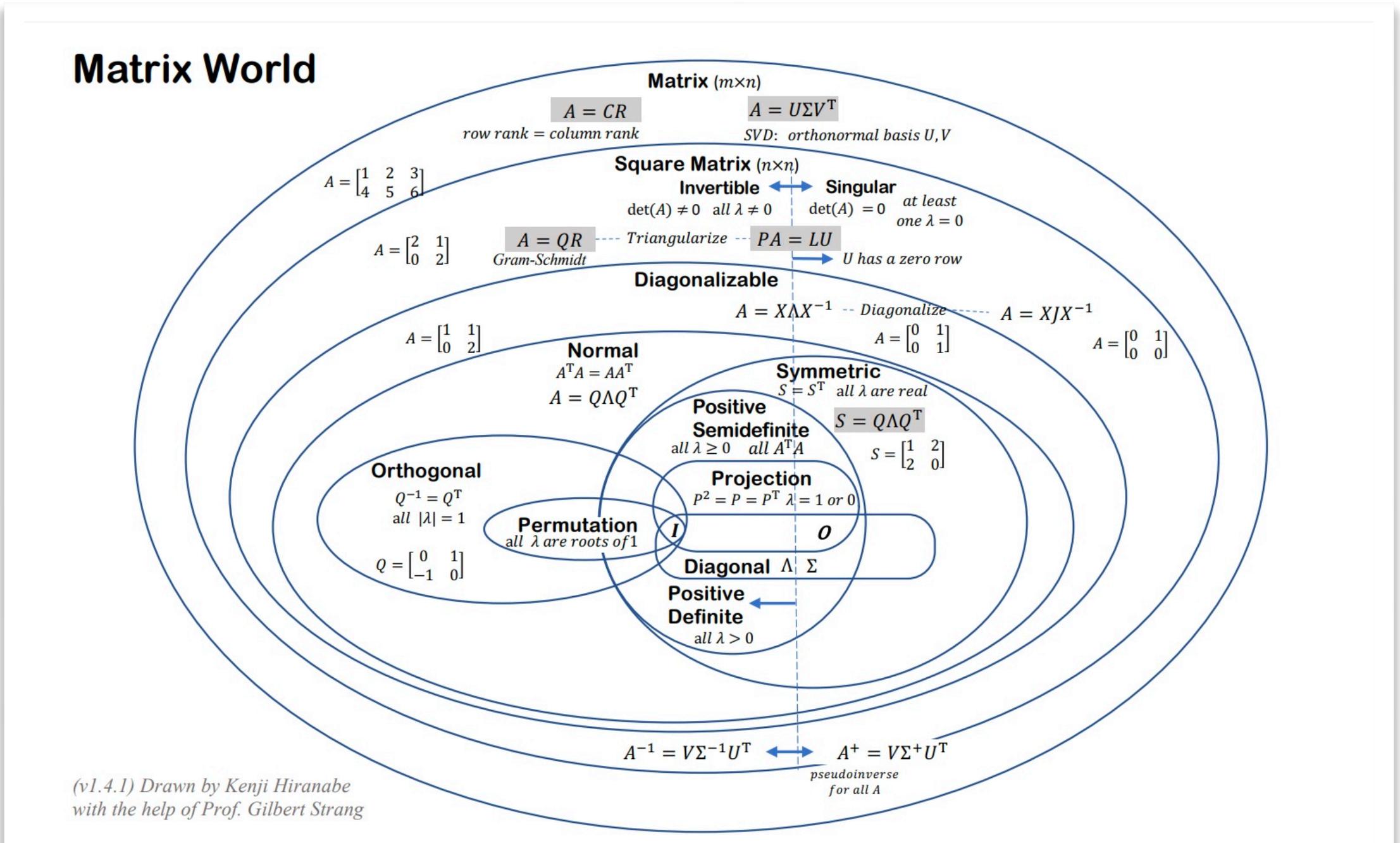




$$A = \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix}$$



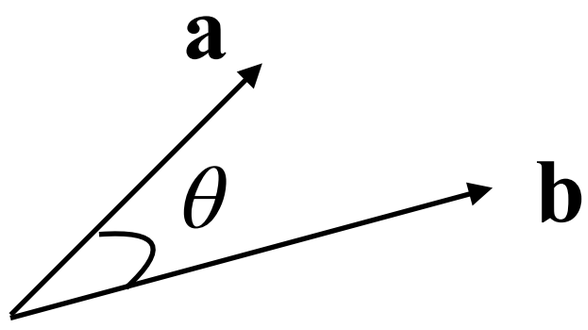
“All” of linear algebra



Background

https://en.wikipedia.org/wiki/Dot_product
https://en.wikipedia.org/wiki/Orthonormal_basis

Dot product:

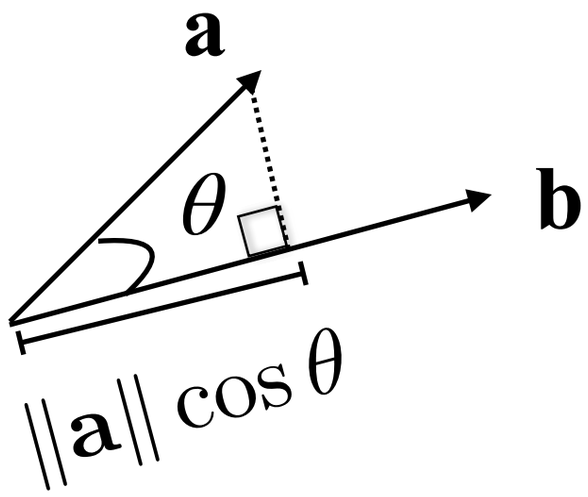


$$\mathbf{a} \cdot \mathbf{b} = \mathbf{a}^T\mathbf{b} = \mathbf{b}^T\mathbf{a} = ||\mathbf{a}||||\mathbf{b}||\cos\theta$$

where $||\mathbf{a}|| = \sqrt{\mathbf{a}^T\mathbf{a}}$

[will sometimes use dot notation and sometimes transpose]

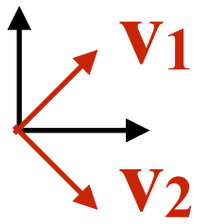
Scalar *projection*:



$$||\mathbf{a}||\cos\theta = \mathbf{a} \cdot \mathbf{b}$$

if $\mathbf{b}$ is a unit vector ($\mathbf{b}^T\mathbf{b} = 1$)

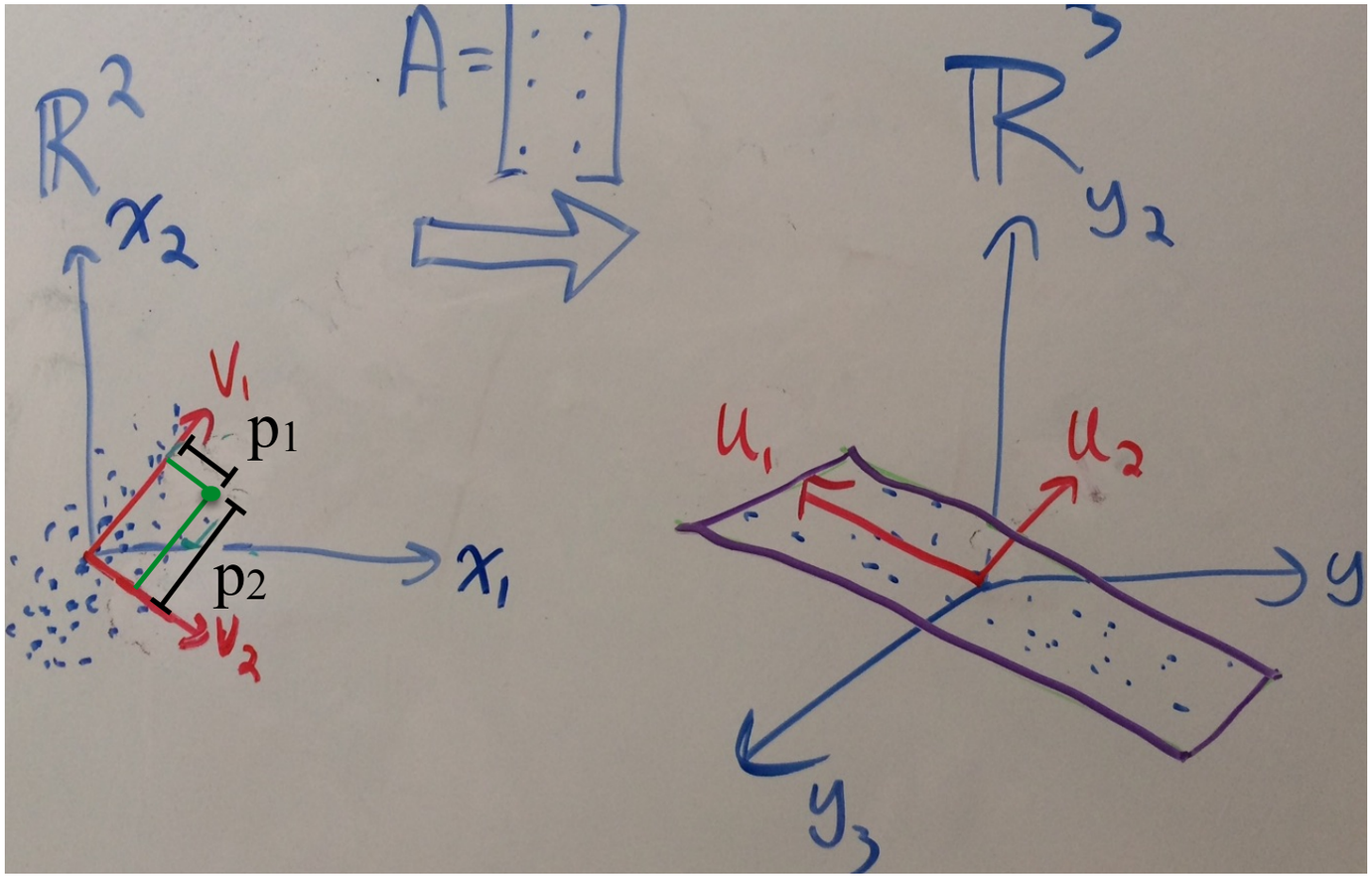
Orthonormal basis:



$$\mathbf{v}_i \in R^n \qquad \begin{matrix}\mathbf{v_i}^T\mathbf{v_i} = 1 \\ \mathbf{v_i}^T\mathbf{v_j} = 0\end{matrix}$$

We can succintly denote orthonormality as

$V^TV = \text{Identity}$ where $V = [\mathbf{v}_1 \,|\, \mathbf{v}_2, \ldots |\, \mathbf{v}_n]$



$$\mathbf{y} = A\mathbf{x}, \quad A \in R^{3\times 2}$$

Let $\mathbf{v_1}$,$\mathbf{v_2}$ be a basis for input space: $\mathbf{v}_1^T\mathbf{v}_1 = \mathbf{v}_2^T\mathbf{v}_2 = 1$, $\mathbf{v}_1^T\mathbf{v}_2 = \mathbf{v}_2^T\mathbf{v}_1 = 0$

Let $\mathbf{u_1}$,$\mathbf{u_2}$,$\mathbf{u_3}$ be a basis for output space: $\mathbf{u}_1^T\mathbf{u}_1 = \mathbf{u}_2^T\mathbf{u}_2 = \mathbf{u}_3^T\mathbf{u}_3 = 1$, $\mathbf{u}_1^T\mathbf{u}_2 = \mathbf{u}_2^T\mathbf{u}_3 = \mathbf{u}_1^T\mathbf{u}_3 = 0$

Project **x** onto input basis:

$$p_1 = \mathbf{v}_1^T\mathbf{x}$$
$$p_2 = \mathbf{v}_2^T\mathbf{x}$$

Scale each coordinate:

$$c_1 = \sigma_1 p_1$$
$$c_2 = \sigma_2 p_2$$

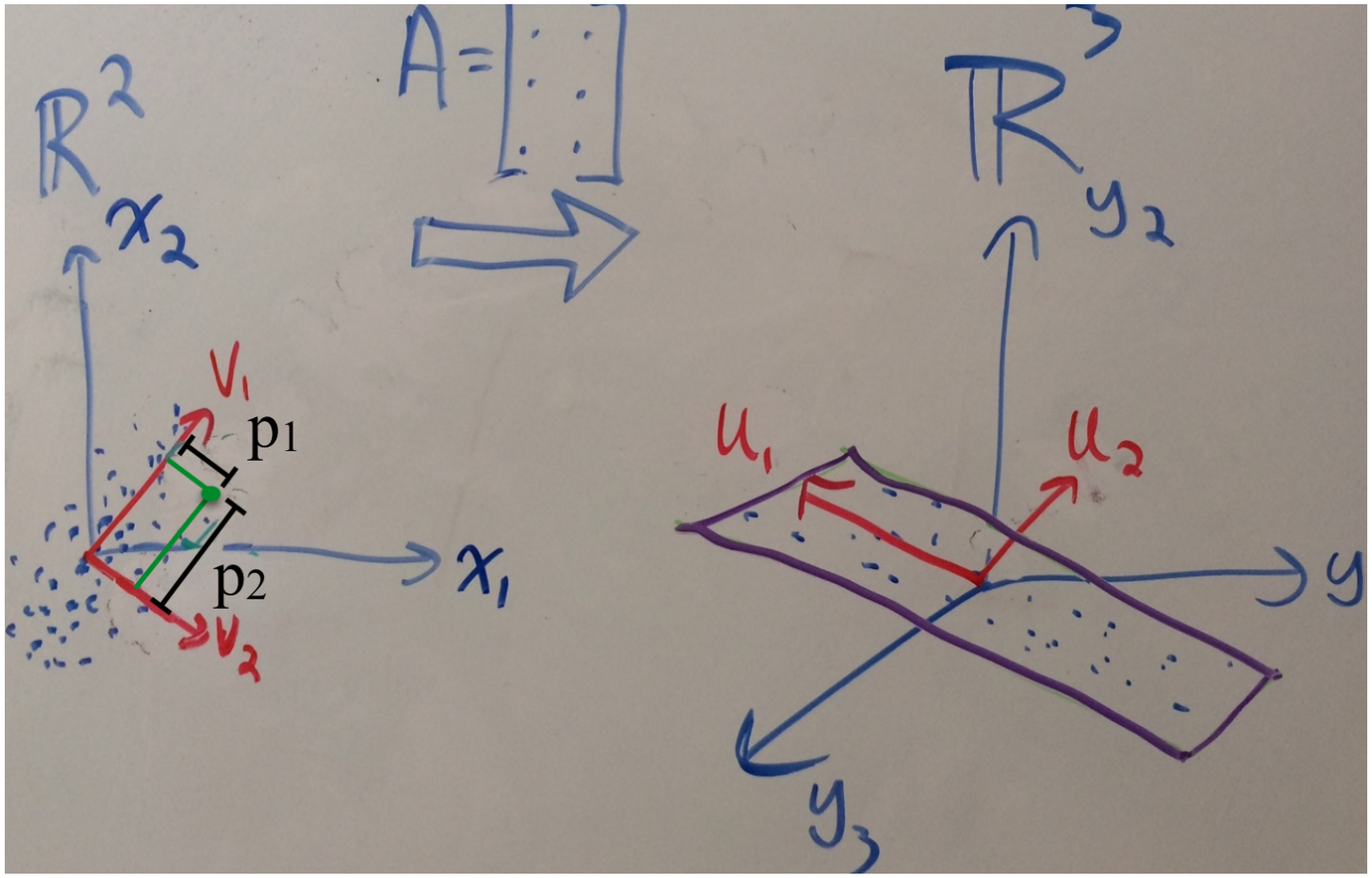
Reconstruct **y** in output space with output basis:

$$\mathbf{y} = c_1\mathbf{u}_1 + c_2\mathbf{u}_2 + c_3\mathbf{u}_3$$
where $c_3 = 0$

That's all there is to a singular value decomposition (SVD)!

σ are singular values, $\mathbf{v}_i$ are right singular vectors, $\mathbf{u}_i$ are left singular vectors

The rest is just writing with compact matrix notation



$$\mathbf{y} = A\mathbf{x}, \quad A \in R^{3\times 2}$$

Let $\mathbf{v_1}$,$\mathbf{v_2}$ be a basis for input space: $\mathbf{v}_1^T\mathbf{v}_1 = \mathbf{v}_2^T\mathbf{v}_2 = 1$, $\mathbf{v}_1^T\mathbf{v}_2 = \mathbf{v}_2^T\mathbf{v}_1 = 0$

Let $\mathbf{u_1}$,$\mathbf{u_2}$,$\mathbf{u_3}$ be a basis for output space: $\mathbf{u}_1^T\mathbf{u}_1 = \mathbf{u}_2^T\mathbf{u}_2 = \mathbf{u}_3^T\mathbf{u}_3 = 1$, $\mathbf{u}_1^T\mathbf{u}_2 = \mathbf{u}_2^T\mathbf{u}_3 = \mathbf{u}_1^T\mathbf{u}_3 = 0$

Project $\mathbf{x}$ onto input basis:

$$p_1 = \mathbf{v}_1^T\mathbf{x}, \quad p_2 = \mathbf{v}_2^T\mathbf{x}$$

$$\begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} -\mathbf{v}_1^T- \\ -\mathbf{v}_2^T- \end{bmatrix} \mathbf{x}$$

Scale each coordinate:

$$c_1 = \sigma_1 p_1, \quad c_2 = \sigma_2 p_2$$

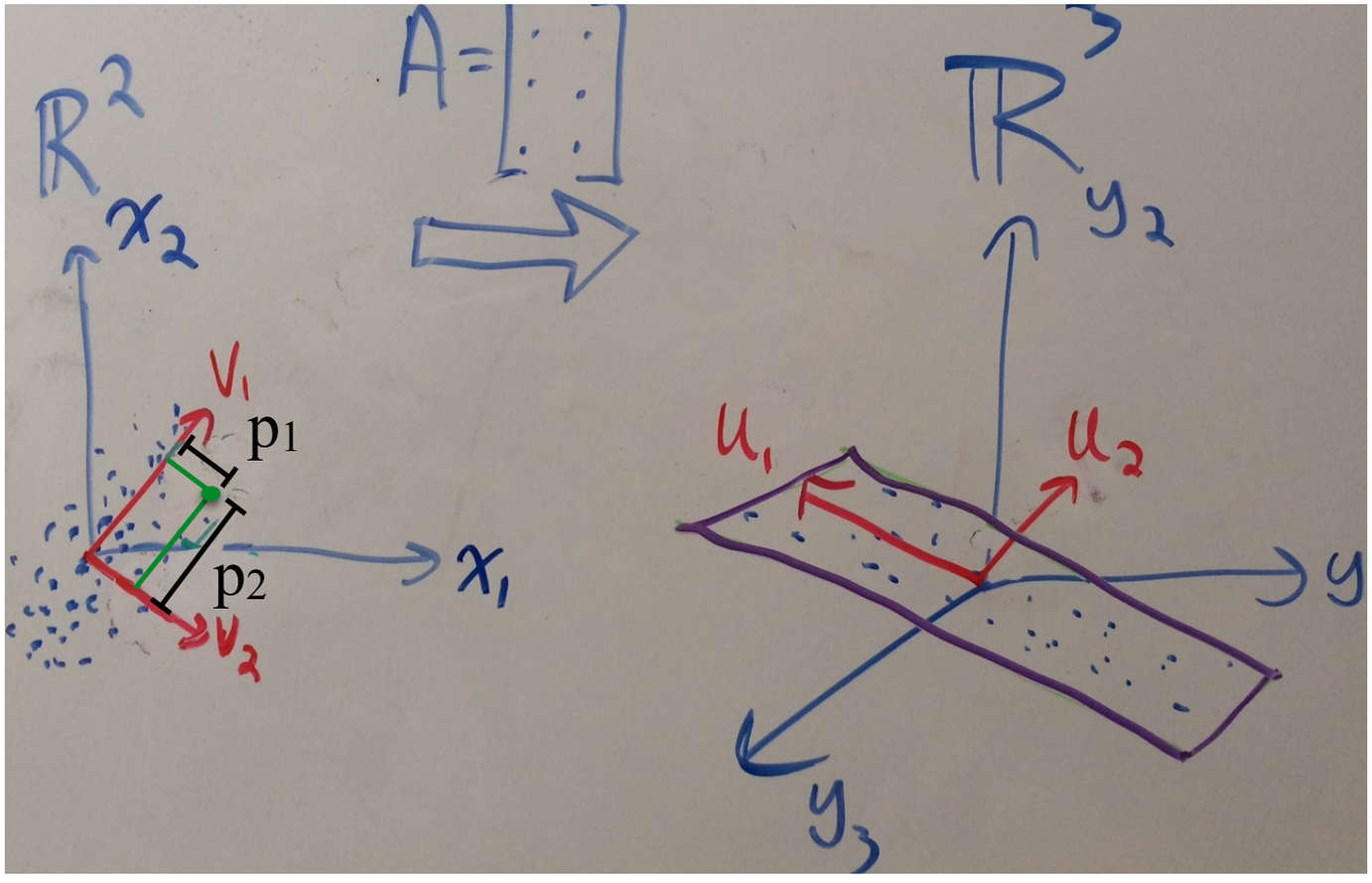
$$\begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \end{bmatrix}$$

Reconstruct $\mathbf{y}$ in output space with output basis:

$$\mathbf{y} = c_1\mathbf{u}_1 + c_2\mathbf{u}_2 + c_3\mathbf{u}_3 \quad \text{where } c_3 = 0$$

$$\mathbf{y} = \begin{bmatrix} | & | & | \\ \mathbf{u}_1 & \mathbf{u}_2 & \mathbf{u}_3 \\ | & | & | \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

Putting it all together…

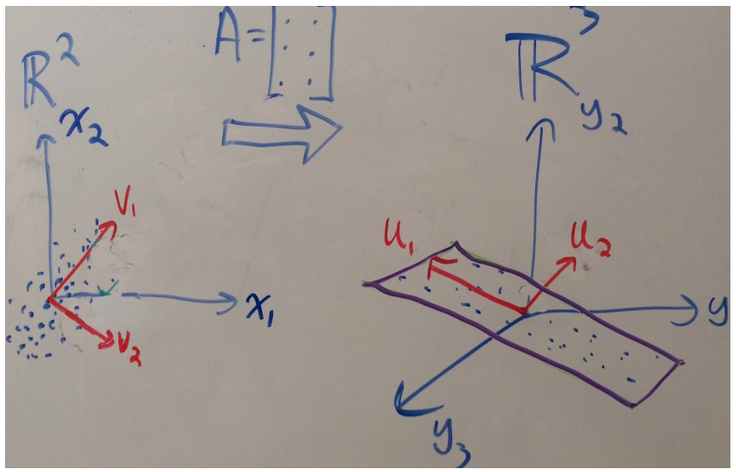


$$\mathbf{y} = A\mathbf{x} = \begin{bmatrix} | & | & | \\ \mathbf{u}_1 & \mathbf{u}_2 & \mathbf{u}_3 \\ | & | & | \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} -\mathbf{v}_1^T- \\ -\mathbf{v}_2^T- \end{bmatrix} \mathbf{x}$$

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

reconstruct scale project

Narrow SVDs



$$A\mathbf{x} = \begin{bmatrix} | & | & | \\ \mathbf{u_1} & \mathbf{u_2} & \mathbf{u_3} \\ | & | & | \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} — & \mathbf{v_1}^T & — \\ — & \mathbf{v_2}^T & — \end{bmatrix} \mathbf{x}$$
$$= U\Sigma V^T\mathbf{x}$$

Can we simplify expression to avoid multiplying by row of all 0s?

"Narrow" SVD: $A = \begin{bmatrix} | & | \\ \mathbf{u_1} & \mathbf{u_2} \\ | & | \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix} \begin{bmatrix} — & \mathbf{v_1}^T & — \\ — & \mathbf{v_2}^T & — \end{bmatrix}$

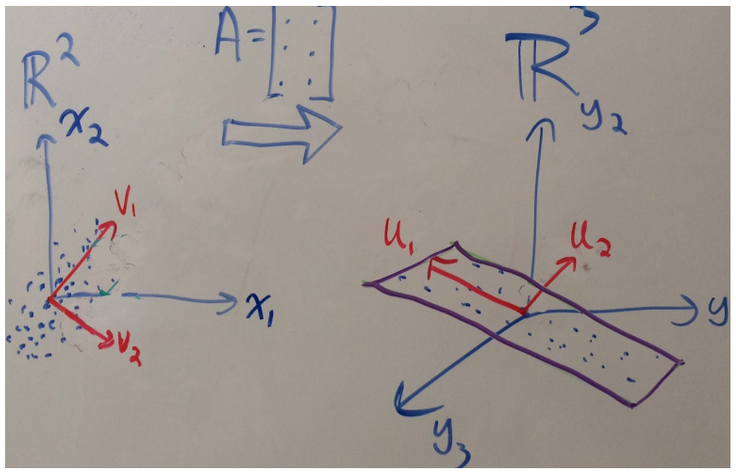
```
# Compute "narrow" SVD with min(m,n) singular values
U, s, Vh = np.linalg.svd(X, full_matrices=False)
print(U.shape, s.shape, Vh.shape)
# Minor price-to-pay: U and Vh may no longer be square
```

What if σ_2 is 0 (or close to it)?

$$A \approx \begin{bmatrix} | \\ \mathbf{u_1} \\ | \end{bmatrix} \sigma_1 \begin{bmatrix} — & \mathbf{v_1}^T & — \end{bmatrix}$$

A can be well approximated by the product of 2 vectors (or rank-1 matrices), which reduces matrix multiplication from O(NM) to O(N+M)

Corollary 0: low-rank approximations



$$A\mathbf{x} = \begin{bmatrix} | & | & | \\ \mathbf{u_1} & \mathbf{u_2} & \mathbf{u_3} \\ | & | & | \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} — & \mathbf{v_1}^T & — \\ — & \mathbf{v_2}^T & — \end{bmatrix} \mathbf{x}$$
$$= U\Sigma V^T\mathbf{x}$$

Best rank-K approximation of A is obtained by using the K largest singular values (and vectors)

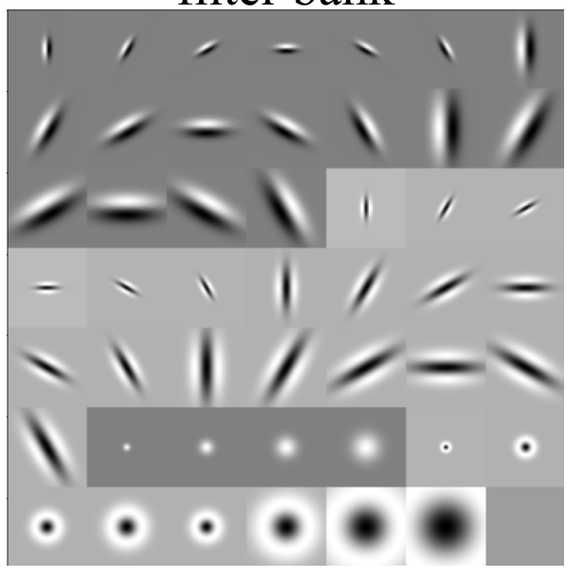
$$\min_{A':rank(A')<k} ||A - A'||_F = U[:, :k]\Sigma[:k, :k]V[:, :k]^T, \quad \text{where} \quad ||A||_F = \sum_{ij} A[i, j]^2$$

Application of SVD: low-rank filter banks

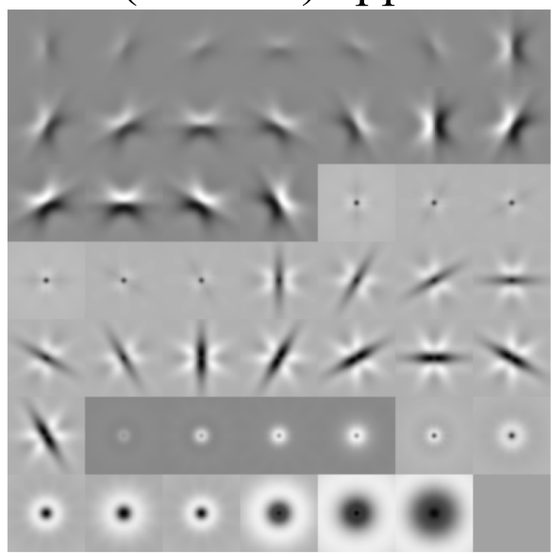
demo_filter_bank.ipynb

Shy & Perona, CVPR94

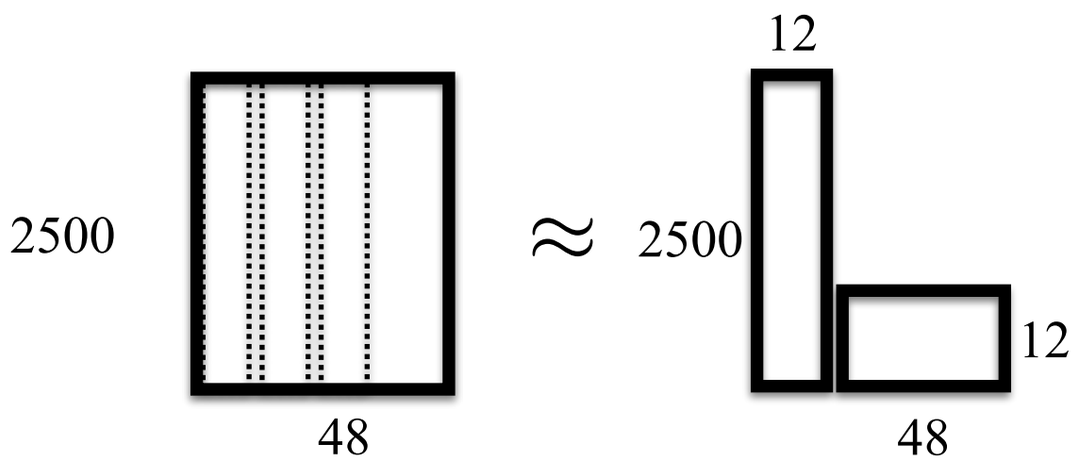
filter bank



low-rank (rank-12) approximation



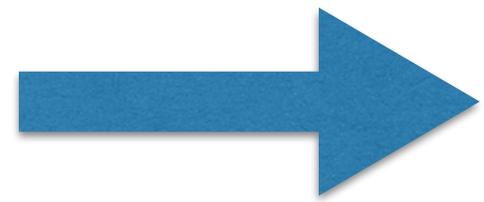
- Column-vectorize each of 48 50x50 filters and stack'em into a 2500x48 matrix
- Apply SVD to generate low-rank approximation of 48 filters as linear combination of 12 *basis* filters



Application of SVD: low-rank filter banks

demo_faces.ipynb

Image dataset



low-rank approximation

Face Recognition Using Eigenfaces

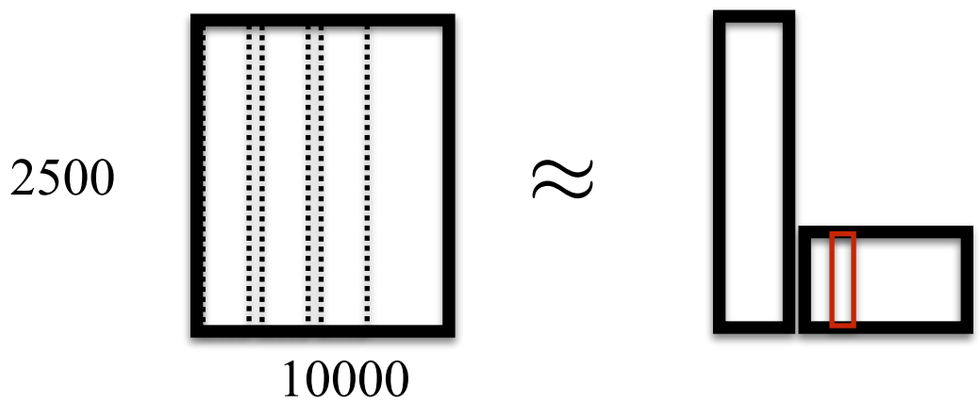
Matthew A. Turk and Alex P. Pentland

Vision and Modeling Group, The Media Laboratory
Massachusetts Institute of Technology

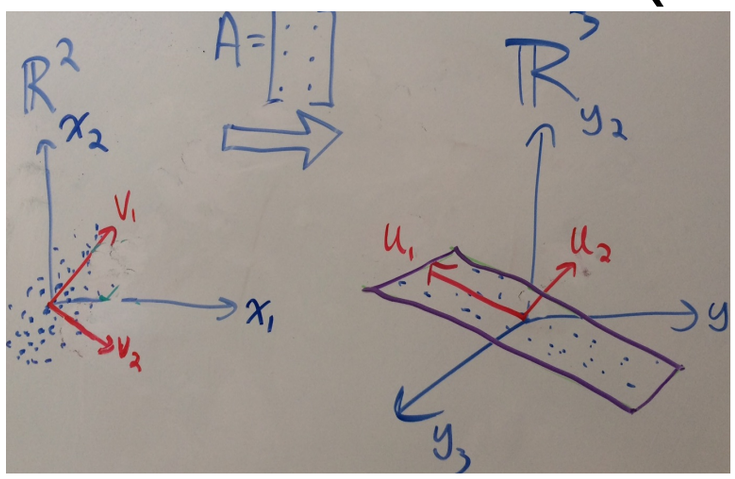
Equivalent to Principal component analysis (PCA), which eigenvectors of A^TA rather than the singular vectors of A. (Next slide)

Aside: how does this compare to K=12 dictionaries that would be learned with K-means clustering all the input image faces?

Here, reconstruction coefficients for each image are dense while K-means coefficients are sparse (all 0's with a single 1)



Corollary 1: positive-semidefinite (PSD) matrices

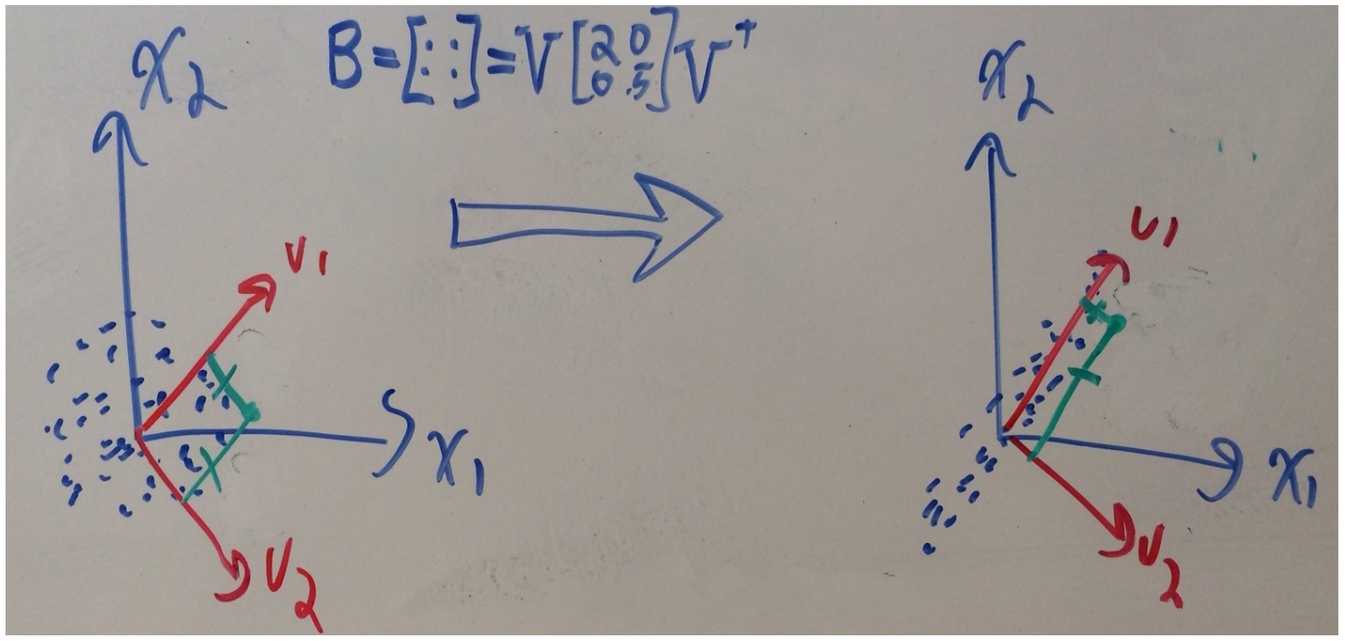


Corollary 1: Positive semi-definite (PSD) matrices. We will encounter many PSD matrices in our class. For example, the Σ covariance matrix in the equation of a 2D Gaussian is PSD: $e^{-\frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)}$. PSD matrices can be defined using a number of properties; perhaps the most common definition is that PSD matrices are square symmetric matrices with non-negative eigenvalues. Personally, I prefer the definition that B is PSD if it can be written as $B = A^TA$ for *some* (possibly non-square) matrix A. This allows me to think of B as a *norm* in the output space - e.g., $||\mathbf{y}||^2 = ||A\mathbf{x}||^2 = \mathbf{x}^TA^TA\mathbf{x} = \mathbf{x}^TB\mathbf{x}$.

PSD as the generalization of the statement "*b* is positive number if $b=a^2$ for some *a*" to a positive *matrix*

Corollary 1: positive-semidefinite (PSD) matrices

$$B = A^T A = V \Sigma^2 V^T = \begin{bmatrix} | & | \\ \mathbf{v}_1 & \mathbf{v}_2 \\ | & | \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \begin{bmatrix} -\mathbf{v}_1^T - \\ -\mathbf{v}_2^T - \end{bmatrix}$$



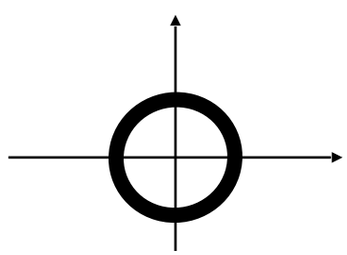
By applying a SVD to A, one can show that B itself can be decomposed as $B = V\Sigma^2 V^T$, where $\mathbf{v_i}$ are eigenvectors with non-negative eigenvalues σ^2. This decomposition is sometimes called a *spectral eigendecomposition*, and can be geometrically viewed as a *projection* onto a rotated basis, *scaling* along that basis, and a *reconstruction* using the same basis vectors. Incidentially, this derivation also suggests two other common definitions of PSD matrices; symmetric matrices such $\mathbf{x}^T B\mathbf{x} \geq 0$ for all $\mathbf{x}$ (since any norm must be greater than or equal to 0) and symmetric matrices who's eigenvalues are all positive or zero (since they can be written as σ^2). Finally, this derivation provides one approach to compute SVD(A); eigendecompose $A^T A$ into V and Σ and solve for U with $A = U\Sigma V^T$.

Alternative visualization of PSD matrices

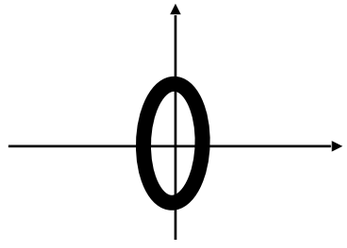
$$B = V\Lambda V^T$$

Consider the set of (x_1, x_2) points for which: $\begin{bmatrix} x_1 & x_2 \end{bmatrix} B \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = 1$

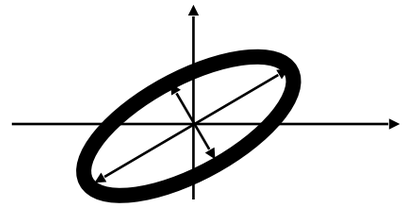
$$\begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = 1, \qquad x_1^2 + x_2^2 = 1$$

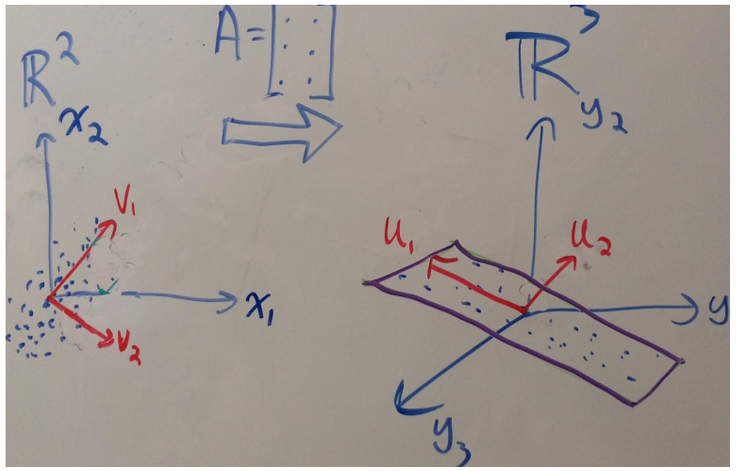


$$\begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = 1, \qquad \lambda_1 x_1^2 + \lambda_2 x_2^2 = 1$$



$$\begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} | & | \\ \mathbf{v}_1 & \mathbf{v}_2 \\ | & | \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \begin{bmatrix} -\mathbf{v}_1^T- \\ -\mathbf{v}_2^T- \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = 1$$





$$A\mathbf{x} = \begin{bmatrix} | & | & | \\ \mathbf{u_1} & \mathbf{u_2} & \mathbf{u_3} \\ | & | & | \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} — & \mathbf{v_1}^T & — \\ — & \mathbf{v_2}^T & — \end{bmatrix} \mathbf{x}$$
$$= U\Sigma V^T \mathbf{x}$$

Corollary 2: Homogenous least squares. We will encounter the following linear equation many times in class:

$$A\mathbf{h} = \mathbf{0}$$

This is equivalent to finding a vector $\mathbf{h}$ in the null space of matrix A. If we are lucky and one of the singular values is 0, then the solution is given by the correponding (right) singular vector. If no singular values are 0, then (intuitively) we should select the singular vector **...** with the smallest singular vector:

$$\min_{\mathbf{h}:\mathbf{h^T h}=1} ||A\mathbf{h}||^2 = V[:,-1]$$

The proof sketch follows by the fact that any input v must project to one of the right singular vectors (because they form a basis). A closely related result is that for any PSD matrix $B = A^T A$, $\min_{\mathbf{h}:\mathbf{h^T h}=1} \mathbf{h^T} B\mathbf{h} = V[:,-1]$, where $V[:,-1]$ the eigenvector with the smallest eigenvalue.

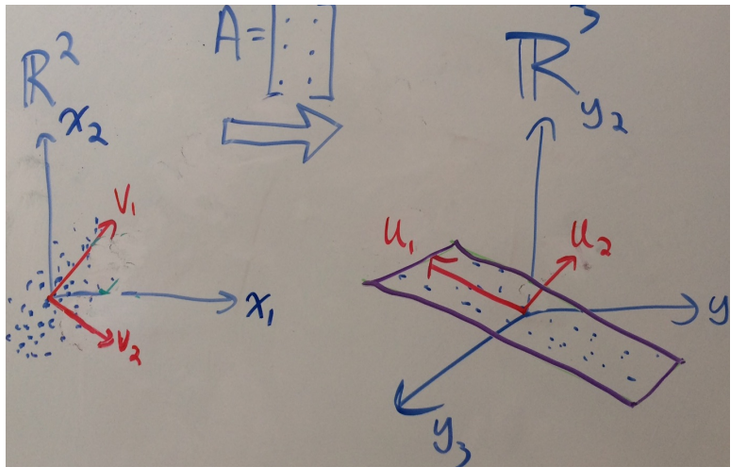
Aside: why does PSD care only about symmetric matrices?

https://www.cs.cmu.edu/~zkolter/course/linalg/linalg_notes.pdf

Note that,

$$x^T Ax = (x^T Ax)^T = x^T A^T x = x^T \left(\frac{1}{2} A + \frac{1}{2} A^T \right) x,$$

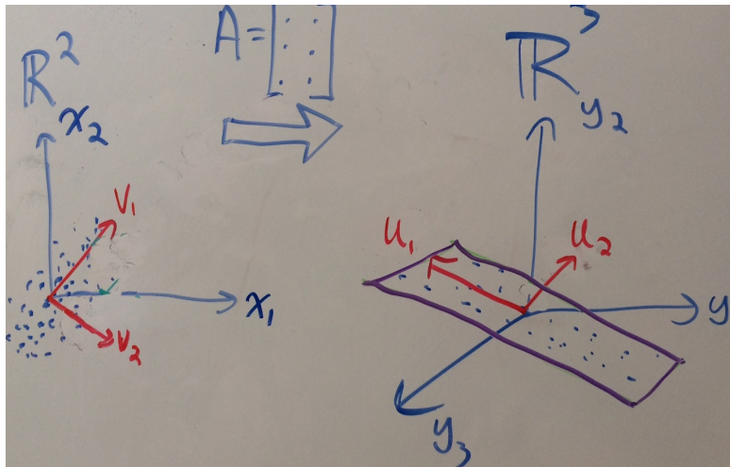
where the first equality follows from the fact that the transpose of a scalar is equal to itself, and the second equality follows from the fact that we are averaging two quantities which are themselves equal. From this, we can conclude that only the symmetric part of A contributes to the quadratic form. For this reason, we often implicitly assume that the matrices appearing in a quadratic form are symmetric.



$$A\mathbf{x} = \begin{bmatrix} | & | & | \\ \mathbf{u_1} & \mathbf{u_2} & \mathbf{u_3} \\ | & | & | \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} — & \mathbf{v_1}^T & — \\ — & \mathbf{v_2}^T & — \end{bmatrix} \mathbf{x}$$
$$= U\Sigma V^T \mathbf{x}$$

Corollary 3: Pseudoinverse. The pseudoinverse of A is given by

$$A^+ = \underset{A^+}{\operatorname{argmin}} ||A^+ A - I||_F =$$

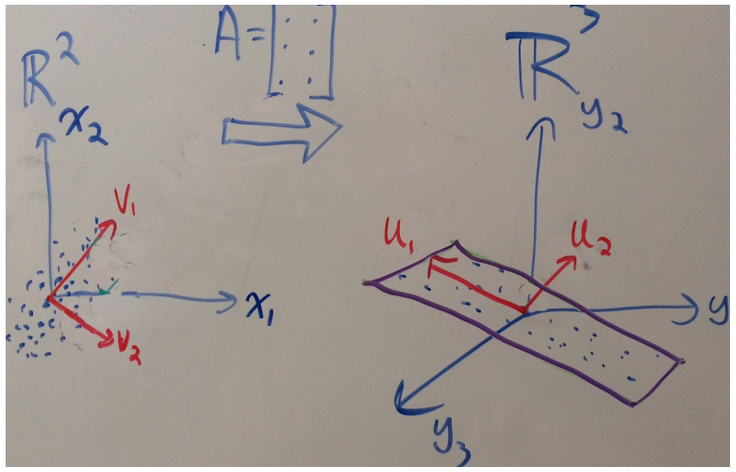


$$A\mathbf{x} = \begin{bmatrix} | & | & | \\ \mathbf{u_1} & \mathbf{u_2} & \mathbf{u_3} \\ | & | & | \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} — & \mathbf{v_1}^T & — \\ — & \mathbf{v_2}^T & — \end{bmatrix} \mathbf{x}$$
$$= U\Sigma V^T\mathbf{x}$$

Corollary 3: Pseudoinverse. The pseudoinverse of A is given by

$$A^+ = \underset{A^+}{\operatorname{argmin}} ||A^+A - I||_F = V \begin{bmatrix} \frac{1}{\sigma_1} & 0 & 0\dots \\ 0 & \frac{1}{\sigma_2} & 0\dots \\ 0 & 0 & \frac{1}{\sigma_3}\dots \\ \vdots & \vdots & \vdots \end{bmatrix}^T U^T$$

which could also be obtained by mimimizing $||AA^+ - I||_F$ (without proof).



$$A\mathbf{x} = \begin{bmatrix} | & | & | \\ \mathbf{u_1} & \mathbf{u_2} & \mathbf{u_3} \\ | & | & | \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} — & \mathbf{v_1}^T & — \\ — & \mathbf{v_2}^T & — \end{bmatrix} \mathbf{x}$$
$$= U\Sigma V^T \mathbf{x}$$

Corollary 4: Least squares. An important special case is the following problem:

$$\min_{\mathbf{x}} ||A\mathbf{x} - \mathbf{b}||^2 \tag{8}$$

(9)

which is solved with the psuedoinverse

$$\mathbf{x} = A^+ \mathbf{b} \tag{10}$$

Note that this holds over overdetermined systems (where A is tall and skinny) and underdetermined systems (where A is short and wide). A particularly common situation is a tall and skinny A with linearly-indepedant columns (which is likely the taller and skinnier A is). Here, the psuedoinverse can be written as

$$A^+ = (A^T A)^{-1} A^T \tag{11}$$

The righthandside of the above expression is also called a *left-inverse*, since $A^+ A = I_{m \times m}$.

Application 1: separable filters

Image of size N^2
Filter of size M^2

Complexity of filtering (let's stick with correlation for now)

$O(N^2M^2)$

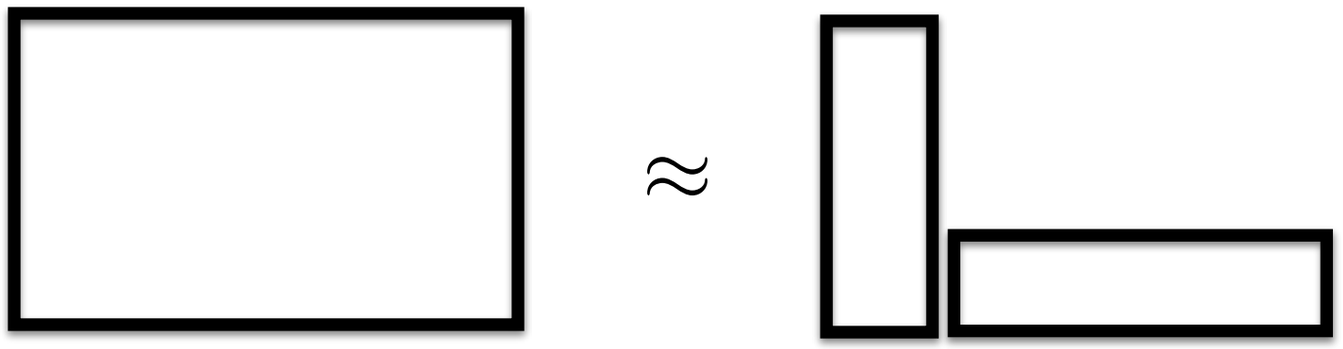
$$H[u,v] = H_x[u]H_y[v]$$

$$G[i,j] = \sum_u \sum_v H[u,v]F[i+u, j+v]$$

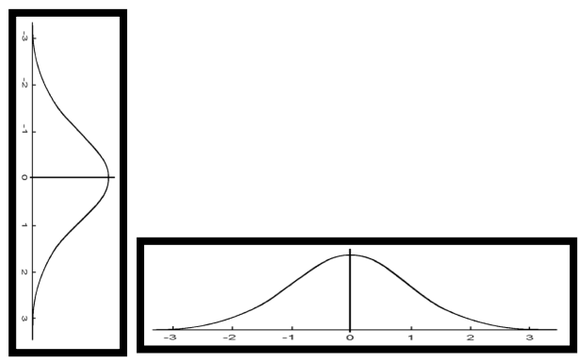
Separability

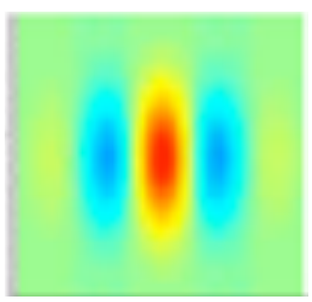
Given a filter, how can we come up with a good separable approximation?

$$H[u, v] \approx H_x[u] H_y[v]$$

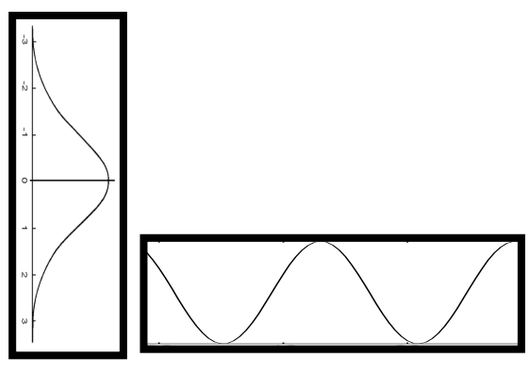


=



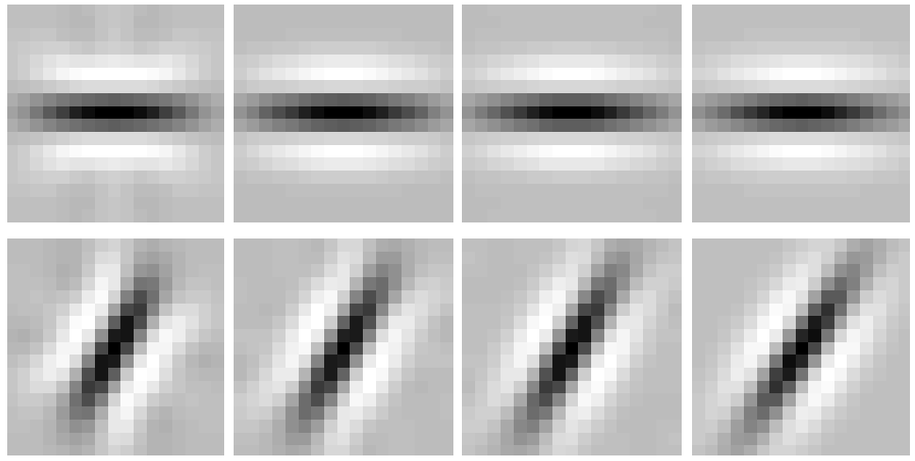


=

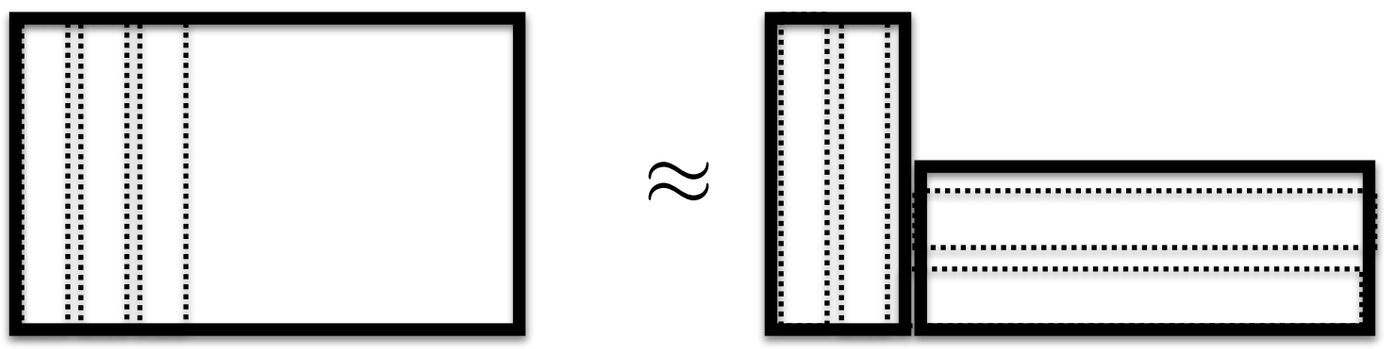


Application 2: eigenfilters

Shy & Perona, CVPR94



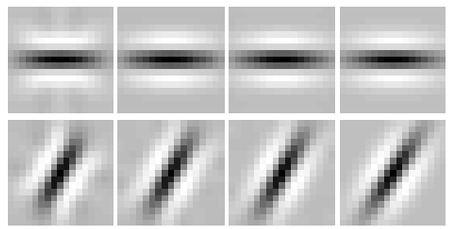
Vectorize each filter and stack each column into a matrix
Apply SVD to generate low-rank approximation



Least-squares method of steerability

Shy & Perona, CVPR94

Rank 1 approximation



$$H[u,v,k] = H_s[u,v]c[k]$$

$$G[i,j,k] = \sum_u \sum_v H[u,v,k]F[i+u,j+v]$$

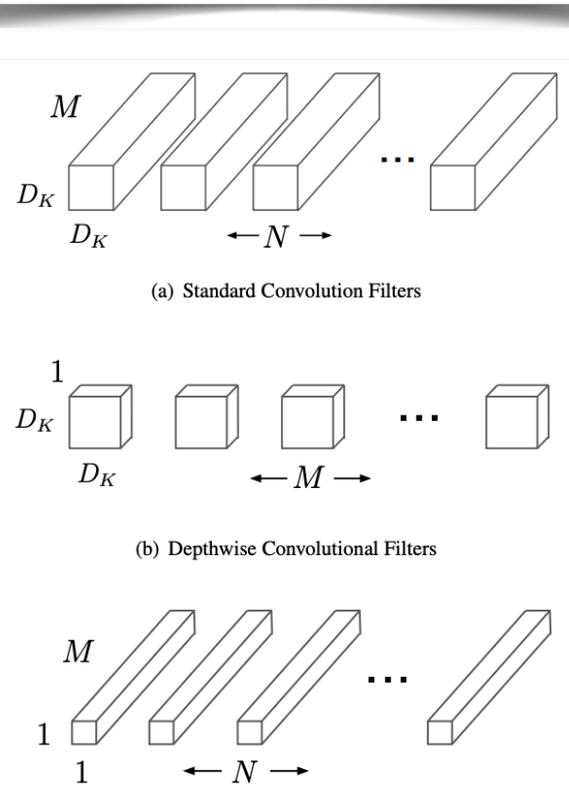
Low-rank separability is ubiquitous in neural networks

30K citations

MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications

Andrew G. Howard Menglong Zhu Bo Chen Dmitry Kalenichenko
Weijun Wang Tobias Weyand Marco Andreetto Hartwig Adam

Google Inc.
{howarda,menglong,bochen,dkalenichenko,weijunw,weyand,anm,hadam}@google.com



(a) Standard Convolution Filters

(b) Depthwise Convolutional Filters

(c) 1×1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

Figure 2. The standard convolutional filters in (a) are replaced by two layers: depthwise convolution in (b) and pointwise convolution in (c) to build a depthwise separable filter.

2K citations

TensoRF: Tensorial Radiance Fields

Anpei Chen[1⋆] Zexiang Xu[2⋆] Andreas Geiger[3] Jingyi Yu[1] Hao Su[4]

[1]ShanghaiTech University [2]Adobe Research
[3]University of Tübingen and MPI-IS, Tübingen [4]UC San Diego
https://apchenstu.github.io/TensoRF/

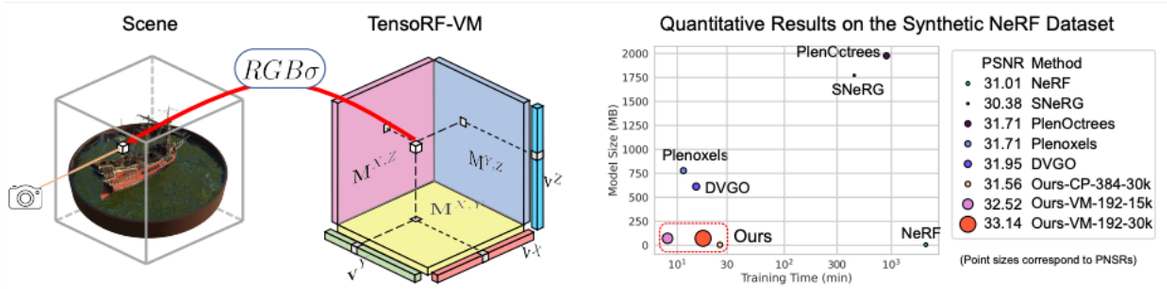
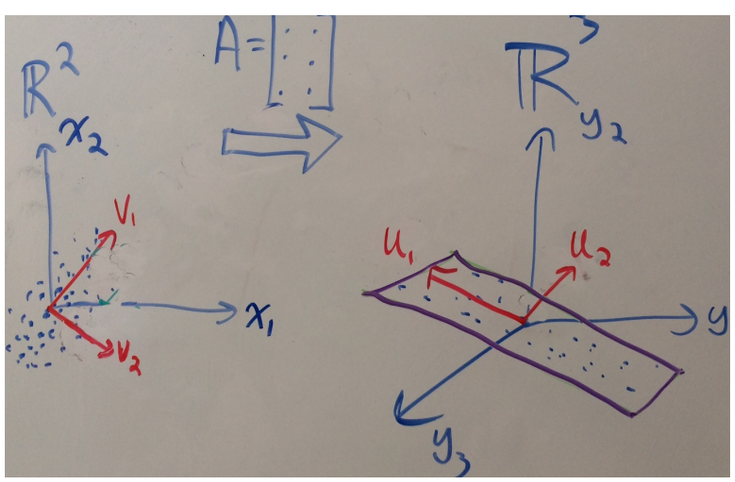
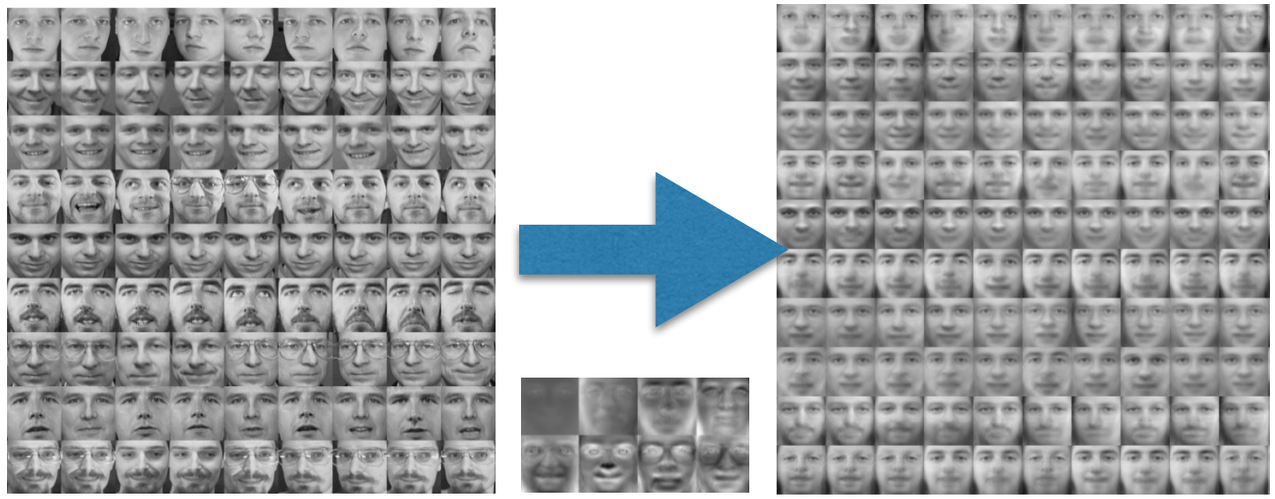


Fig. 1: Left: We model a scene as a tensorial radiance field using a set of vectors ($\mathbf{v}$) and matrices ($\mathbf{M}$) that describe scene appearance and geometry along their corresponding axes. These vector/matrix factors are used to compute volume density σ and view-dependent RGB color via vector-matrix outer products for

A look back: singular value decompositions





- Low-rank approximations
- Positive semidefinite (PSD) matrices (and spectral decomposition)
- (Homogenous) least squares
- Pseudo-inverse