

# Computational Vision

U. Minn. Psy 5036  
Daniel Kersten  
Lecture 10: Image processing

## Initialize

### Read in Add-in packages:

```
In[168]:= Off[General::"spell1"];  
SetOptions[ArrayPlot, ColorFunction -> "GrayTones",  
  DataReversed -> True, Frame -> False, AspectRatio -> Automatic,  
  Mesh -> False, PixelConstrained -> True, ImageSize -> Small];
```

### The input 64x64 image: face

```
In[170]:= face = ImageData[ImageReflect[ColorConvert[
```

```
{width, height} = Dimensions[face];  
gface = ArrayPlot[face]
```

Out[172]=



---

## Upcoming dates:

Final project outlines due: Nov. 13th.

### Final projects

Format

Should be written like a scientific paper.

Might require most of the code to be put in appendices.

Can use modules you find elsewhere, but preserve copyrights, and reference

Your "audience" will be your class peers.

## Review: Multiresolution, spatial filter models of early visual processing

### Single-channel spatial filtering

One assumes a convolution filter of a fixed shape (e.g. the weights of a DOG filter) that gets applied across the whole image to predict an output "neural image".

images can be represented in the fourier domain in terms of: amplitude and phase spectra.

convolution of an image with a filter in the space domain is mathematically equivalent to:

$$\text{image} \otimes \text{kernel} = F^{-1}(F(\text{image}) \times F(\text{kernel}))$$

or using Mathematica functions:

$$\text{InverseFourier}[\text{Fourier}[\text{filter}] * \text{Fourier}[\text{image}]]$$

convolution is used to model: blur, neural "images", the linear stage in neural networks, deep convolutional neural networks

### Multiple spatial frequency channels

Most neurons in primary visual cortex, in particular "simple cells" and "complex cells", have receptive fields whose sizes are limited, and are selective for particular spatial frequencies and orientations. This suggests that neurons in primary visual cortex can be grouped into "channels" consisting of population of neurons across retinotopic locations, but with similar frequency and orientation tuning. (See classic experiment of Campbell and Robson, 1968)

To model how these simple cell population transform a pattern of incoming image intensities, one assumes a set of convolution filters, each set has a fixed shape (e.g. the weights of gabor filters describe a fixed template) that get applied across the image to predict a set of output "neural images". Each neural image represents information at a particular spatial scale and orientation. The collection of filters, identical except for position, is called a channel.

Global vs. local:

Sinusoidal basis functions are global filters. Global filtering is not a good model because they give up spatial localization, at the expense of spatial frequency information (the filters would have to extend across the whole space implying large receptive fields).

## Multiresolution analysis with local filters

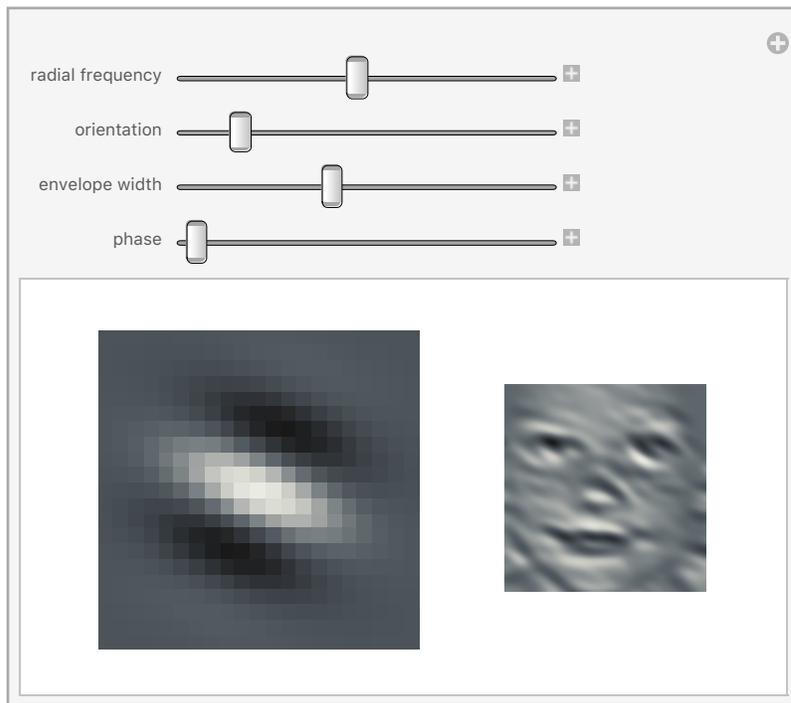
Instead of (global) sinusoidal basis functions, the convolutions use localized "gabor function" filters:

```
In[174]:= Clear[Grating, GratingPatch, kern];
```

```
In[175]:= Grating[x_,y_,fx_,fy_,phase_] := Cos[(2.0 Pi (fx x + fy y) + phase)];
GratingPatch[x_,y_,fx_,fy_,sig_,phase_] := Exp[-((x)^2 + (y)^2)/(2*sig^2)]*Grating[x,y,fx
kern[fx_, fy_, sig,phase_] :=
  Table[GratingPatch[x, y, fx, fy, sig,phase], {x, -1, 1, .1}, {y, -1, 1, .1}];
```

The following demo illustrates a single neural image for various choices of spatial frequency and orientation. In addition, the envelope width manipulates "bandwidth"--i.e. more cycles under the gabor means narrow spatial frequency tuning and thus narrow bandwidth. The phase slider moves one from "bar" to "edge" type receptive fields.

```
In[178]:= Manipulate[
  GraphicsRow[{
    ArrayPlot[kern[fr * Cos[theta], fr * Sin[theta], sig, phase]], ArrayPlot[
      ListConvolve[kern[fr * Cos[theta], fr * Sin[theta], sig, phase], face]]},
    {{fr, 1, "radial frequency"}, .1, 2}, {{theta, .4, "orientation"}, 0, Pi},
    {{sig, .4, "envelope width"}, .001, 1}, {{phase, 0, "phase"}, .0, Pi / 2}]
```



Out[178]=

- ▶ 1. If you wanted to design a simple template to detect (rather than identify) faces based on convolution with a gabor filter, what filter parameters might you use?

```
In[179]:= zebra = ImageData[
  ImageReflect[ColorConvert[RemoveAlphaChannel[

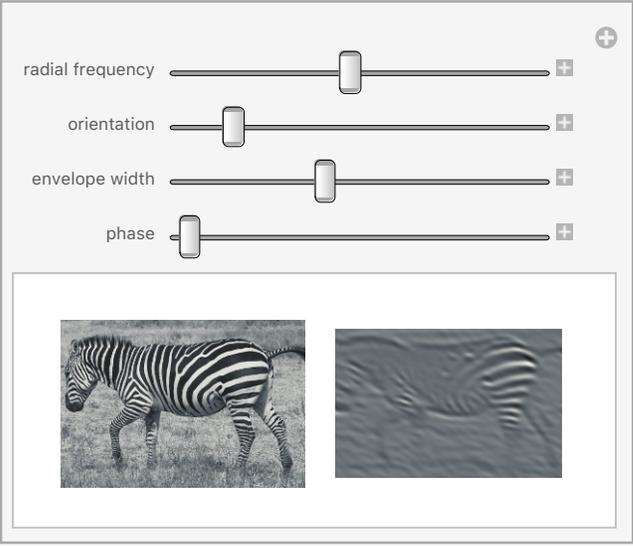
```

```
Dimensions[
  zebra]
```

```
Out[180]= {174, 255}
```

```
In[181]:= Manipulate[
  GraphicsRow[{
    ArrayPlot[zebra], ArrayPlot[
      ListConvolve[kern[fr * Cos[theta], fr * Sin[theta], sig, phase], zebra]]},
    {{fr, 1, "radial frequency"}, .1, 2}, {{theta, .4, "orientation"}, 0, Pi},
    {{sig, .4, "envelope width"}, .001, 1}, {{phase, 0, "phase"}, .0, Pi / 2}]
```

Out[181]=



## Self-similarity

When the filters have the same shape except for a change of scale ( $x \rightarrow \alpha x$ ), where  $\alpha$  is a constant, they are called self-similar. The self-similar idea is important to vision because of the importance of scale sensitive and scale invariant processing. Further, the self-similar aspect of neural filter models bears a close resemblance to the emerging mathematical field of wavelet analysis.

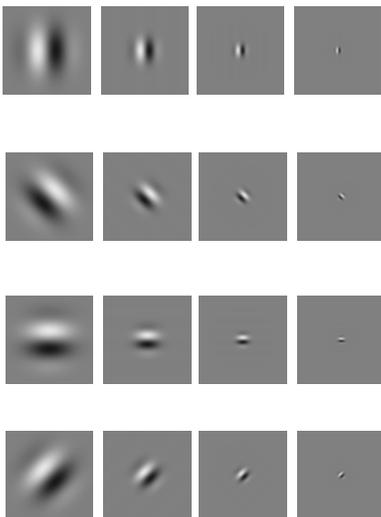
## Human statistical efficiency for detecting gabor patches

Burgess, Wagner, Jennings and Barlow (1981) combined the SKE observer and spatial frequency analysis of human vision to find out which patterns human observers had the best contrast discrimination efficiency. They showed in a 1981 Science article that narrowly windowed sinusoids were detected with high efficiency (>70%) when added to static visual noise. These targets were detected more efficiently than simple disks of light.

You have all the tools to replicate the experiment of Burgess et al. You can compute  $d'$  for the ideal observer for signal-known-exactly patterns. And you can generate Gaussian-windowed sinusoids and add them to gaussian white noise. If you measure the percent correct, and convert that to  $d'$  for the human observer, you can calculate the absolute efficiency for human detection--and contribute to answer the question of what the eye sees best. Watson, Barlow & Robson (1983) found that that a 7 c/deg grating drifting at 4 Hz, (with a narrow gaussian envelope in space and time) was detected more efficiently than other patterns. They did not add artificial noise. They used photopic light levels, and taking into account the photon noise, they calculated the quantum efficiency at less than  $<0.05\%$ .

### Bottom-line: image coding in terms of scale and orientation: A model for human spatial image representation

At each spatial location, project the image onto a collection of basis vectors (i.e. compute the dot product) that span a range of spatial scales and orientations:



In general, these neural models of basis functions may be over-complete, and non-orthogonal. And there may be a range of phases. Above we show only the "sine-phase" or "edge-detectors" of Hubel and Wiesel.

### How linear are V1 neurons?

There are kinds of non-linearities that have been introduced to refine, or model other types of neural populations: 1) rectification--neuronal firing rate is by definition non-negative. The combination of on-center, off-center responses can be treated as a theoretical unit to represent negative and positive signal values. 2) adding squared outputs of sine-phase and cosine-phase filter to produce "contrast energy" filters; 3) Contrast normalization, where the output of an otherwise linear neuron is normalized by the "energy" outputs of ones nearby in space. We'll discuss some of these in later lectures

## What is a multiresolution scale/orientation representation good for?

What is the computational significance of a wavelet-like decomposition?

Analysis of what vision needs to recognize objects, etc..

- > First stages of local feature extraction
  - Edges, bars, ...over various spatial scales
- > “front end” for subsequent processing
  - textures, curvature, corners, ...

Efficient coding?

- > savings in neurons, or metabolic requirements?
- > representations for efficient learning, subsequent coding?
- > analysis of natural image statistics

## Image processing tools

### Mathematica has a library of built-in functions for doing image manipulations

See the Basic Image Manipulation, Image Processing & Analysis, and the Image Filtering & Neighborhood Processing guides.

Here are some examples:

```
In[41]:= Image[Reverse[face]]
```



We've seen a simple blur using BoxMatrix:

```
In[42]:= ImageConvolve[, BoxMatrix[2] / 9] // ImageAdjust ;
```

```
In[43]:= BoxMatrix[2] / 9
```

```
Out[43]= {{1/9, 1/9, 1/9, 1/9, 1/9}, {1/9, 1/9, 1/9, 1/9, 1/9}},
          {{1/9, 1/9, 1/9, 1/9, 1/9}, {1/9, 1/9, 1/9, 1/9, 1/9}, {1/9, 1/9, 1/9, 1/9, 1/9}}
```

And the built-in Laplacian of a Gaussian filter which can be used to model center-surround, lateral inhibitory filtering:

In[44]:=

```
ifiltered = ImageAdjust[LaplacianGaussianFilter[, 2]]
```



Out[44]=



In[45]:= ImageAdjust[ifiltered]

Out[45]=



- ▶ 2. What does ImageAdjust do? Try ListPlot[ImageData[ifiltered][[32]]] with and without using ImageAdjust

We can quickly generate the “neural images” for on- and off-center responses:

```
In[46]:= GraphicsRow[{ifiltered, ImageMultiply[Binarize[ifiltered], ifiltered],  
ImageMultiply[ColorNegate[Binarize[ifiltered]], ifiltered]}]
```

Out[46]=



- ▶ 3. The above used a default threshold to separate above mean from below mean responses. Try using:  $\text{threshold} = \text{Mean}[\text{Mean}[\text{ImageData}[\text{ifiltered}]]]$  as the second argument of Binarize[].

## Point operations

### Various contrast definitions

We've seen that for simple stimuli, contrast can be defined as:

$(I_{\max} - I_{\min}) / (I_{\max} + I_{\min})$ : Called "Michelson contrast". Particularly appropriate for gratings, or stimuli with primary luminance peak and valleys in the image.

$\Delta I / I_{\text{background}}$ : Often used in psychophysics of small points/disks against a larger background, or temporal increments/decrements ( $\Delta I$ ) relative to a base level ( $I_{\text{background}}$ ).

$\Delta I / I_{\text{mean}}$ : Gives same number as Michelson for gratings, when  $\Delta I$  corresponds with the amplitude of the grating.

For complex stimuli, we could represent contrast in terms of standard statistical measures on luminance. Let the mean luminance:  $I_{\text{mean}} = \sum_{x,y} I(x,y) / N$ , where N is the number of pixels. The variance of

the luminance is:  $\frac{\sum_{x,y} (I(x,y) - I_{\text{mean}})^2}{N}$ . Then the standard deviation could be used to provide an overall measure of contrast:

$$\sqrt{\frac{\sum_{x,y} (I(x,y) - I_{\text{mean}})^2}{N}}$$

(The formula for the unbiased variance estimate could also be used, where  $N$  is replaced by  $N-1$ ). However, the above standard deviation definition depends on the units (e.g. luminance is measured in candelas/meter<sup>2</sup>). Further, it is useful to have contrast definitions that come closer to capturing the perceptual aspects of contrast in an image.

This is the rationale for defining contrast at a point--called local contrast. The visual system is relatively insensitive to the mean luminance, suggesting that a useful measure of local contrast is luminance divided by the mean, whose ratio is dimensionless. In addition, the subjective, qualitative difference between "bright" and "dark" suggests positive and negative contrast, respectively. Local contrast at a point  $(x,y)$  as:

$$c(x,y) = \frac{I(x,y) - I_{\text{mean}}}{I_{\text{mean}}}$$

The mean of  $c(x,y)$  is zero by definition, with positive and negative values corresponding to bright and dark relative to the mean. (Note that local contrast can be defined as function of time too,  $c(x,y,t)$ .)

The definition of local contrast at a point raises the question: over what range should  $I_{\text{mean}}$  be calculated? This question is relevant to the problems of local adaptation to light level, and to tone mapping. The default for us will be to take the mean over the whole image.

Given  $c(x,y)$ , we can calculate a summary measure of contrast. The variance of  $c(x,y)$  is the same as the variance of  $I(x,y)/I_{\text{mean}}$  and is given by:  $\frac{\sum_{x,y} c^2(x,y)}{N}$ . The root mean square (r.m.s.) of set of measurements

$m_i^2$  is by definition:  $\sqrt{\frac{\sum_i m_i^2}{N}}$ . Thus one can define r.m.s. contrast as the square root of the variance of  $c$ :

$$\sqrt{\frac{\sum_{x,y} c^2(x,y)}{N}}$$

This definition of r.m.s. contrast provides us with a useful summary measure of overall "contrastiness" for a complex image.

By analogy with physics (substituting space for time) the square of r.m.s contrast is sometimes called "contrast power". And then "contrast energy" is defined as: contrast power x area. In psychophysics, "area" is often measured in squared degrees of visual angle. Contrast energy is a useful measure when one is concerned about how the size (or duration) of an image patch affects its visibility. If  $x$  and  $y$  are measured in degrees of visual angle, then,  $\int c^2(x, y) dx dy$  is the contrast energy, which in the discrete case can be estimated as:

$\sum_{x,y} c^2(x, y) \times \left(\frac{\text{area}}{N}\right) = \sum_{x,y} c^2(x, y) \Delta x \Delta y$ , where  $\Delta x$  and  $\Delta y$  are the dimensions of a pixel in degrees.

Recall from Lecture 5:  $d' = \frac{\text{r.m.s. contrast} \times \sqrt{N}}{\sigma}$

If we extend the above contrast definitions to time, in terms of general appearance, the apparent contrast of an image doesn't increase with time, so contrast energy (contrast power x duration) is irrelevant. However, over short durations (less than 100 msec or so), human vision approximately integrates contrast power, so contrast energy is a better predictor of threshold. But for longer durations, contrast power is the better predictor.

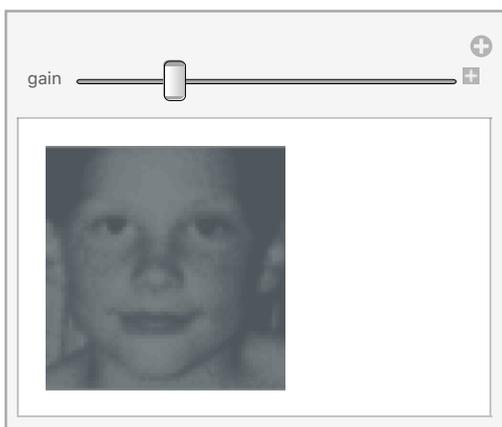
- ▶ 4. Modify the definition of contrast at a point to have a variable argument for the size (s) of a local spatial region over which to compute the average. Use `Manipulate[]` to take as input an image whose values range from 0 to 1, and as output an image in which the output is proportional to the contrast computed over the restricted range sxs.
  - Side note on the relation of contrast energy of an image to its fourier representation: **Parseval's theorem** says that the contrast energy of an image is equal to the integral of the square of the amplitude spectrum (called the power spectrum) over spatial frequency. *Contrast energy doesn't depend on the phase relationships between the frequency components.*

## Contrast manipulations

Adjusting contrast (gain=1 leaves image unchanged, gain=0 reduces it to a uniform field):

```
In[182]:=  $\mu = \text{Mean}[\text{Flatten}[\text{face}]]$ ; gain = 0.045;
Manipulate[
ArrayPlot[gain (face -  $\mu$ ) +  $\mu$ , Mesh  $\rightarrow$  False,
Frame  $\rightarrow$  False, PlotRange  $\rightarrow$  {Min[face], Max[face]}],
{{gain, 0.045}, 0, 1}]
```

Out[183]=



## Psychophysics and contrast

When measuring human visual sensitivity, it is important to carefully measure and calibrate the image stimuli. Because standard 8-bit computer displays resolve 256 graylevels, it can be useful to convert the

stimuli into a range going from 0 to 255. Scale so values are represented as graylevels between 0 and 255:

```
In[49]:=  $\alpha = 255 / (\text{Max}[\text{face}] - \text{Min}[\text{face}]);$   

 $\beta = -\alpha \text{Min}[\text{face}];$ 
```

```
In[51]:=  $\text{face256} = \alpha \text{face} + \beta;$ 
```

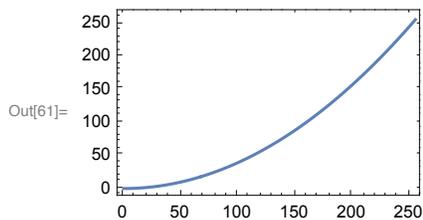
- ▶ 5. Exercise: Normalize “face” so that it has a mean level of zero, and an r.m.s. contrast of 1. Use ListPlot and Flatten[] to show a scatter plot of the values before and after the scaling.
- ▶ 6. Exercise: Given that human contrast sensitivity for a sinewave grating can be as high as 500, could you get a good measure of it using a typical computer graphics screen? (answer: for fine measurements of human contrast sensitivity, 8 bits of intensity is too coarse. 10 to 12 bits is better.)
- ▶ 7. Exercise: Produce a negative image by reversing the contrast

## Gamma correction for displays

Computer operating systems allow one to adjust for various non-linearities between displays. A typical screen has a non-linear relationship between measured screen intensity and the voltage supplied. A traditional way of summarizing the non-linear relationship is in terms of "gamma": intensity = a x voltage<sup>gamma</sup>. Let's assume that we are using intensity units that range from 0 to 255 and voltage units also going from 0 to 255. intensity = 255<sup>(1-gamma)</sup> x voltage<sup>gamma</sup>:

```
In[60]:=  $\text{output}[\text{input}_, \text{gamma}_] := 255^{1-\text{gamma}} \text{input}^{\text{gamma}};$   

Plot[output[x, 2], {x, 0, 255}, Frame → True, ImageSize → Small]
```



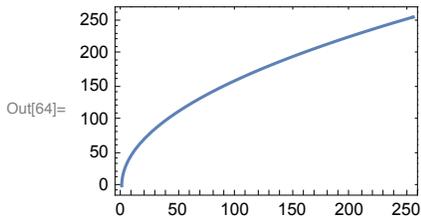
The computer's display card has a look-up-table (LUT) that can be loaded with the inverse gamma function to linearize the display. The LUT remaps input values to output voltages so that there is a linear relationship between internal values and screen luminance.

```
In[62]:= Solve[output == 255 ^ (1 - gamma) input ^ gamma, input]
```

**Solve:** Inverse functions are being used by Solve, so some solutions may not be found; use Reduce for complete solution information.

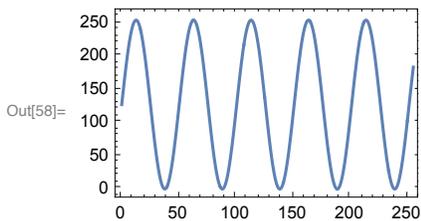
```
Out[62]= { { input → (255-1+gamma output)1/gamma } }
```

```
In[63]:= inverse[output_, gamma_] := (255-1+gamma output)1/gamma;
Plot[inverse[x, 2], {x, 0, 255}, Frame → True, ImageSize → Small]
```

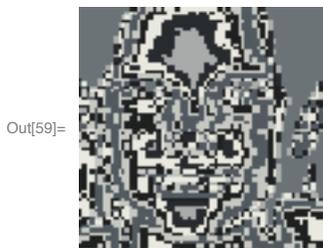


One can do silly things too, like a many-to-one mapping:

```
In[57]:= (output2[input_] := 127. Sin[ $\frac{\text{input}}{8}$ ] + 127;);
Plot[output2[x], {x, 0, 255}, Frame → True, ImageSize → Small]
```



```
In[59]:= ArrayPlot[output2[face256], PlotRange → {0, 255}]
```

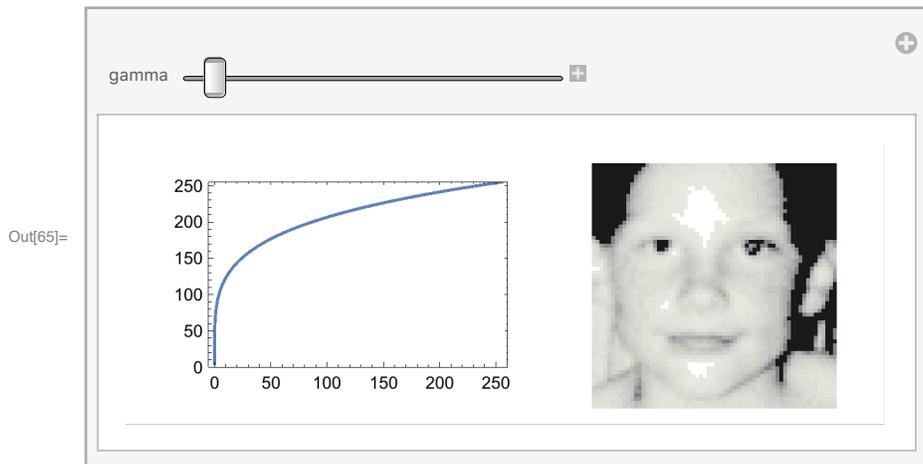


..but maybe this isn't so silly. What does the fact that you can still see a recognizable face-like form in the above picture tell you about human vision?

### Using gamma to do point operations

You can also use the gamma transformation to do non-linear point operations on an image to concentrate some intensity values more than others.

```
In[65]:= Manipulate[
GraphicsRow[{Plot[output[x, gamma], {x, 0, 255},
Frame → True, ImageSize → Small, PlotRange → {0, 255}],
ArrayPlot[output[face256, gamma], PlotRange → {0, 255}]}],
{gamma,
0.1,
4}]
```

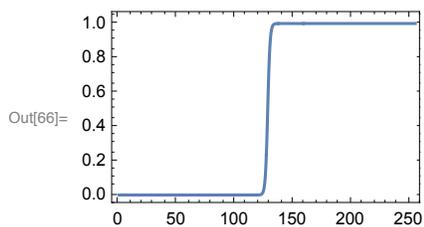


## Sigmoidal contrast manipulation

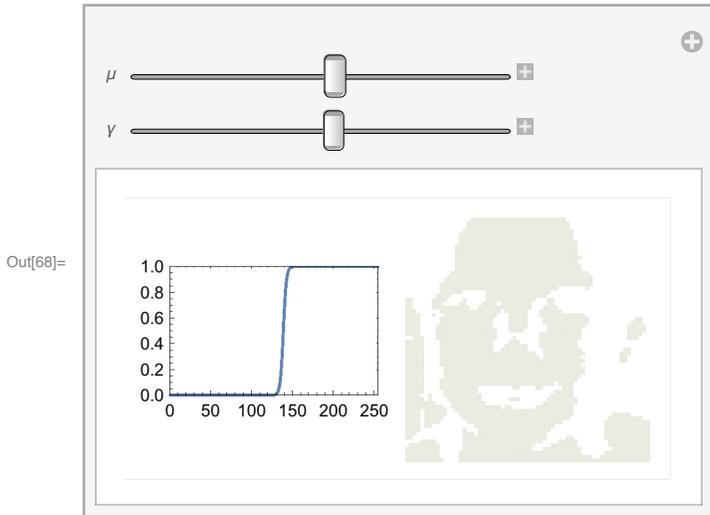
Here is a gain function (called the "logistic function") that manipulates contrast smoothly--a "soft" threshold:

```
In[66]:= squash[x_, μ_, γ_] := N[ $\frac{1}{1 + e^{-\gamma(x-\mu)}}$ ];
```

```
Plot[squash[x, 128, 1], {x, 0, 255}, Frame → True, ImageSize → Small]
```



```
In[67]:= gain = 0.045` ;  $\mu_0$  = Mean[Flatten[face256]] ;
Manipulate[GraphicsRow[
  {Plot[squash[x,  $\mu$ ,  $\gamma$ ], {x, 0, 255}, Frame  $\rightarrow$  True, PlotRange  $\rightarrow$  {{0, 255}, {0, 1}},
  ArrayPlot[squash[(face256 -  $\mu_0$ ) +  $\mu_0$ ,  $\mu$ ,  $\gamma$ ], PlotRange  $\rightarrow$  {Min[face], Max[face]}]}],
  {{ $\mu$ , 128}, 0, 255}, {{ $\gamma$ , .5}, 0, 1}]
```



As  $\gamma$  grows, the sigmoid approaches a hard-threshold. Images that are forced to have only two values are called "Mooney images".

We can make Mooney images more directly using a function that takes an image and sets pixels bigger than  $\tau$  to 255, and if less than (or equal to)  $\tau$ , to 0:

```
In[75]:= Mooney[image_,  $\tau$ _] := Map[If[# >  $\tau$ , 255, 0] &, image, {2}];
ImageReflect[Image[Mooney[face256, 32]]]
```



See also: `Binarize[]`, `ColorQuantize[]`. And the famous "Dalmation dog illusion" in perception books, or via a web search.

See Moore and Engel, 2001, and Hegd  J & Kersten D. (2007) for applications of Mooney images to studying human vision.

Gray levels are effectively quantized at low light levels. Mooney images mimic this effect but at high light levels.

- 8. Exercise: Write a function that quantizes an image to a set of gray levels specified by a set of thresholds:  $\tau_1, \tau_2, \tau_3, \dots, \tau_{N-1}$ . Set  $N=3$ . (Try using `Which[]`).

## Simple statistics

First-order, i.e. don't take into account relations between pixels. We've already seen a summary measure of overall "contrastiness".

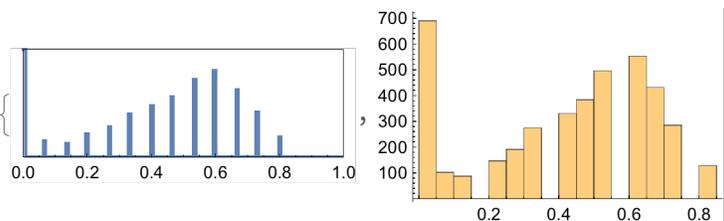
```
In[77]:=  $\mu$  = Mean[Flatten[face]];
 $\sigma$  = Sqrt[Variance[Flatten[face]]];
rmscontrast = Sqrt[Variance[Flatten[face]] / Mean[Flatten[face]]]

Out[79]= 0.600059
```

## Histograms

For images or for any data:

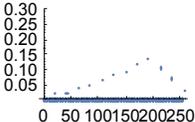
```
In[80]:= {ImageHistogram[, Histogram[Flatten[face]]}
```

Out[80]= 

You can tell that the image is quantized at a coarse level (less than 4 bits). Alternatively, you could calculate the histogram with more basic functions. To do the pattern match below, the floating point numbers are first converted to integers using `Round[]`. If we normalize the histogram so that the sum is one, then we have a probability:

```
In[82]:= domain = Range[0, 255];
Freq = Map[Count[Round[Flatten[face256]], #] &, domain];

In[84]:= ListPlot[ $\frac{\text{Freq}}{\text{Plus@@Freq}}$ , PlotStyle -> PointSize[0.02],
PlotRange -> {0, .3}, ImageSize -> Tiny]
```

Out[84]= 

## Getting regions of images

Three ways to pull out a region:

```
ArrayPlot[Take[face256, {1, 32}, {1, 64}]]
```



Or to get rows 1 to 32 and columns 1 to 64 you can use:

```
ArrayPlot[face256[[1 ;; 32, 1 ;; 64]]]
```



The above method is particularly useful to learn because it generalizes to general lists of lists, and has parallels in both python/numpy and matlab (see “slicing” in python).

We can also use the built-in image function ImageTake[] to get rows 32 through 64:

```
ImageTake[, {32, 64}]
```



- ▶ 9. Use Manipulate[] and ImageTake[] to make an interactive selection tool for picking out rectangular regions of arbitrary size

### Getting coordinates by hand

```
ArrayPlot[face256, PixelConstrained -> {1, 1}]
```



Click on the image above to select it. Now bring up the Drawing Tools, under the Graphics menu. Now use the “cross hairs” tool to select the first the lower left point{x0,y0}, and then the upper right point {x1,y1} as the corners of the rectangular patch that you want. Copy and paste in the argument of the Round[] cell below. Here are coordinates for diagonal points for the eyes.

```
In[100]:= Round[ToExpression@{"10.09", "26.71"}, {"36.1", "48.99"}];
Reverse[Transpose[%]];
ArrayPlot[Take[face256, %[[1]], %[[2]]]]
```

Out[102]=



Or using Mathematica's built-in graphics utility:

Click out the output of Image[]. This will bring up a whole set of image processing tools

In[103]:=

```
Image[
```



Out[103]=




---

## Geometric image manipulations using function interpolation

You can turn a discrete representation into a continuous one using ListInterpolation[]. Then one can do symbolic operations such as take a derivative, or do morphing.

```
In[104]:= faceFunction =
  ListInterpolation[Transpose[face], {{-1, 1}, {-1, 1}}, InterpolationOrder → 1];
DensityPlot[faceFunction[x, y], {x, -1, 1}, {y, -1, 1}, PlotPoints → 256,
  Mesh → False, AspectRatio → Automatic, Frame → None, ColorFunction → "GrayTones"]
```



Now you can calculate intensities “between pixels”:

```
In[109]:= faceFunction[.1, .23]
```

```
Out[109]= 0.444778
```

- ▶ 10. Compare the plots with `InterpolationOrder → 0` and `InterpolationOrder → 1`.

## Morphing

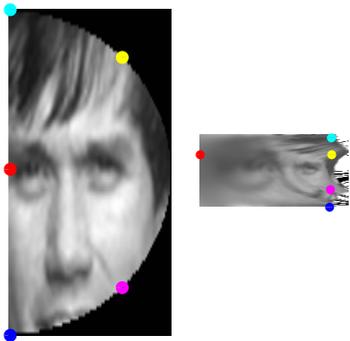
```
In[107]:= faceFunction = ListInterpolation[Transpose[face], {{-1, 1}, {-1, 1}}];
```

```
In[108]:= DensityPlot[faceFunction[Sign[x] x^2, Sign[y] y^2], {x, -1, 1},
  {y, -1, 1}, PlotPoints -> 100, Mesh -> False, AspectRatio -> Automatic,
  Frame -> None, ColorFunction -> "GrayTones", ImageSize -> Small]
```

Out[108]=



- FINAL project idea: How does our ability to recognize objects degrade with geometrical deformations?
- FINAL project idea: Use Mathematica's Interpolation functions to model the log polar model of the retinotopy of primary visual cortex to show how the foveal representation gets expanded on cortex.



## More filtering: Calculating the spatial gradient of an image using function interpolation

Both first and second spatial derivatives have been used to model edge detection/amplification by neural processes in primary visual cortex.

Hubel and Wiesel's "edge detector" can be interpreted as an approximation to a first order derivative ("gradient" in 2D, see  $\nabla f$ ), and their "bar detector" as a second derivative (see  $\nabla^2 G$  below).

In this section, we show how to use symbolic derivatives to find the gradient.

The gradient (or gradient vector field) of an image intensity function  $f$ , is given by the vector  $\nabla f$ .

$$\vec{\nabla}f = \left( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right)$$

The vector length is  $|\nabla f|$ . At a given point, it has a maximum length in the direction of greatest change in intensity.

$$|\nabla f| = \left| \left( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right) \right| = \sqrt{\left( \frac{\partial f}{\partial x} \right)^2 + \left( \frac{\partial f}{\partial y} \right)^2}$$

Let `filterface = f`, where we've blurred out face a little to reduce quantization artifacts:

```
In[136]:= kernel = {{1, 1, 1}, {1, 1, 1}, {1, 1, 1}};
(*Alternatively, you could use BoxFilter[]*)
filterface = ListConvolve[kernel, face];

In[138]:= faceFunction = ListInterpolation[Transpose[filterface], {{-1, 1}, {-1, 1}}];

In[139]:= Clear[x, y]

In[140]:= nx[x_, y_] := Evaluate[D[faceFunction[x, y], x]];
ny[x_, y_] := Evaluate[D[faceFunction[x, y], y]];
ImageGradient[x_, y_] := Evaluate[Sqrt[D[nx[x, y], x]^2 + D[ny[x, y], y]^2] ];
```

Plot the rate of change in the x-direction, where intensity in the output image is proportional to the rate of change.

```
In[143]:= temp = Table[nx[x, y], {x, -1, 1, .005}, {y, -1, 1, .005}];
ArrayPlot[Transpose[temp]]
```

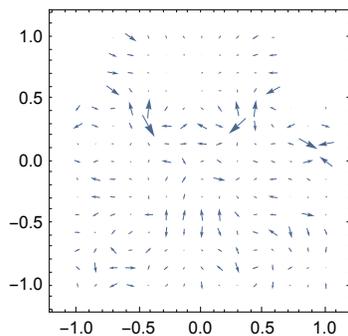
Out[144]=



You can use `VectorPlot` to visualize the gradient field:

```
In[145]:= VectorPlot[{nx[x, y], ny[x, y]}, {x, -1, 1}, {y, -1, 1}, ImageSize -> Small]
```

Out[145]=



Plot the magnitude of the gradient to highlight regions of the image where contrast is changing the

most rapidly:

```
In[146]:= DensityPlot[ImageGradient[x, y], {x, -1, 1}, {y, -1, 1}, PlotPoints → width,
  Mesh → False, Frame → False, ColorFunction → "Rainbow", ImageSize → Small]
```

Out[146]=



## Discrete spatial derivative filtering with built-in image functions

Use can also use image specific filters. `GradientFilter[image,r]` gives an image corresponding to the magnitude of the gradient of image, using discrete derivatives of a Gaussian of pixel radius  $r$ .

```
In[147]:= GradientFilter[, 6] // ImageAdjust
```

Out[147]=



The  $\nabla^2 G$  operator (one of the forms used to represent a “mexican hat” filter) in the previous lecture first convolves the image with a Gaussian blur kernel, and then takes the Laplacian  $\nabla^2$ . This filter effectively blurs an image before taking the derivatives.

Later we’ll see why when we study edge detection.

```
In[148]:= LaplacianGaussianFilter[, 2] // ImageAdjust
```

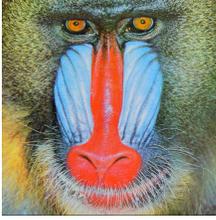
Out[148]=




---

## Manipulating color images

```
image = ExampleData[{"TestImage", "Mandrill"}]
```



```

RGBvalues = ImageData[image];
Dimensions[RGBvalues]
{512, 512, 3}

{512, 512, 3}
{512, 512, 3}

{512, 512, 3}
{512, 512, 3}

{512, 512, 3}
{512, 512, 3}

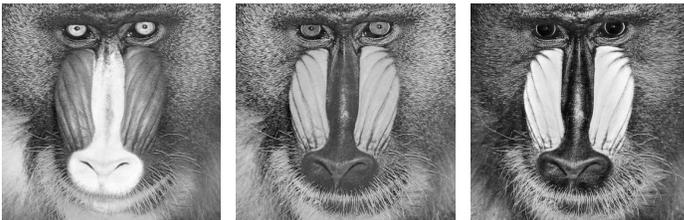
{512, 512, 3}
{512, 512, 3}

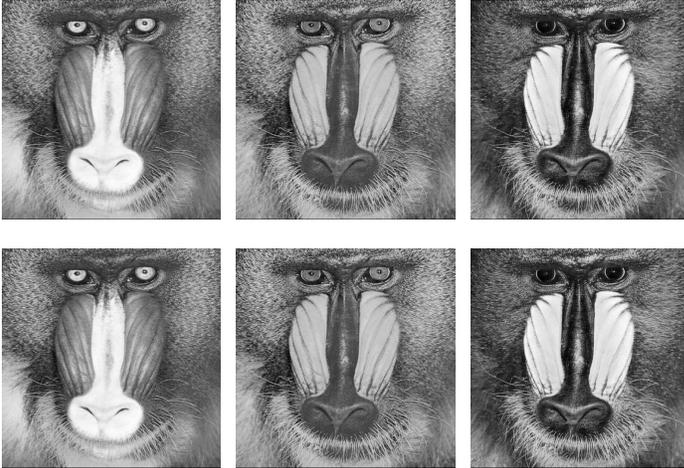
{512, 512, 3}
{512, 512, 3}

reds = Map[#[[1]] &, N[RGBvalues], {2}];
greens = Map[#[[2]] &, N[RGBvalues], {2}];
blues = Map[#[[3]] &, N[RGBvalues], {2}];

GraphicsRow[{Image[reds], Image[greens], Image[blues]}]

```





You can also use the function:

```
ColorSeparate[];
```

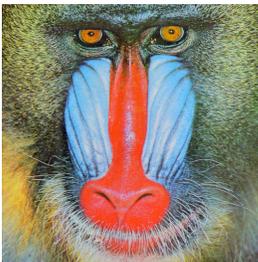
**A weighted average of RGB values to produce a luminance image:**

```
grayscale = Map[ $\frac{0.3 \#[[1]] + 0.59 \#[[2]] + 0.11 \#[[3]]}{255}$  &, N[RGBvalues], {2}];
Image[grayscale] // ImageAdjust;
```

Note the weights above are arbitrary, and the chosen values will depend on the color calibration.

**Putting the R, G, B images back together:**

```
r = reds;
g = greens;
b = blues;
temp2 =
  Partition[Transpose[{Flatten[r], Flatten[g], Flatten[b]}], Dimensions[r][[2]]];
Image[
  temp2]
```



---

## Next time

Efficient coding

---

## References

- Adelson, E. H., Simoncelli, E., & Hingorani, R. (1987). Orthogonal Pyramid Transforms for Image Coding. Paper presented at the Proc. SPIE - Visual Communication & Image Proc. II, Cambridge, MA.
- Barlow, H. B., & Olshausen, B. A. (2004). Convergent evidence for the visual analysis of optic flow through anisotropic attenuation of high spatial frequencies. *J Vis*, 4(6), 415-426.
- Daugman, J. G. (1988). An information-theoretic view of analog representation in striate cortex, *Computational Neuroscience*. Cambridge, Massachusetts: M.I.T. Press.
- Engel, S. A., Glover, G. H., & Wandell, B. A. (1997). Retinotopic organization in human visual cortex and the spatial precision of functional MRI. *Cereb Cortex*, 7(2), 181-192.
- Gold, J. M., Murray, R. F., Bennett, P. J., & Sekuler, A. B. (2000). Deriving behavioural receptive fields for visually completed contours. *Curr Biol*, 10(11), 663-666.
- Hegd e, J., & Kersten, D. (2010). A Link between Visual Disambiguation and Visual Memory. *Journal of Neuroscience*, 30(45), 15124-15133. doi:10.1523/JNEUROSCI.4415-09.2010
- Konishi, S. M., Yuille, A. L., Coughlan, J. M., & Zhu, S. C. (2003). Statistical edge detection: Learning and evaluating edge cues. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(1), 57-74.
- Olman, C. A., & Kersten, D. (2004). Classification objects, ideal observers & generative models. *Cognitive Science*, 28, 227-239.
- Moore, C., & Engel, S. A. (2001). Neural response to perception of volume in the lateral occipital complex. *Neuron*, 29(1), 277-286.
- Schwartz, E. L. (1980). A quantitative model of the functional architecture of human striate cortex with application to visual illusion and cortical texture analysis. *Biol Cybern*, 37(2), 63-76.
- <http://library.wolfram.com/howtos/images/#histograms>