Correlation in Facial Recognition

Correlation in Facial Recognition

Size

(m*n) x 5

Expression 1

This matrix is 5x5, and is computed as $A^T \times A$ where A is the matrix formed by dividing each column by the mean of that column and by the standard deviation of that column.

Expression 2

This matrix is $(m * n) \times (m * n)$ and is calculated as $A \times A^T$, where A uses the same definition as above. This is because ten pixels all very near black should be treated as having high correlation, even though the standard deviation of just those five is low enough to amplify the noise. The giant matrix produced contains the correlation between every pair of pixels in the frame of the input images.

Symmetry

Left Right symmetry in human faces would create high correlation values between pixels that exist on the opposite side of the same face. This would manifest itself as high values in The area near the principal diagonal, representing locations on the same or nearby rows, but mirrored across the vertical centerline.

Test your understanding...

```
SetDirectory[NotebookDirectory[]];
<< ../Imports.m
```

Correlation in Faces

```
namesTable[]
```

```
fivefaces = images[[{57, 65, 89, 113, 321}]];
Map[Image, fivefaces]
```

```
resampledfaces = Map[ArrayResample[#, {50, 40}] &, fivefaces];
Dimensions[resampledfaces]
Map[Image, resampledfaces]
```

```
{5, 50, 40}
```

```
flattened = Transpose @ Flatten[resampledfaces, {2, 3}];
Dimensions[flattened]
```

```
{5, 2000}
```

This 5x5 matrix shows which images are most correlated with eachother.

```
correlation1 = correlation[Transpose @ flattened] // MatrixForm
```

$$\begin{pmatrix} 1. & 0.439795 & -0.0030219 & 0.044081 & 0.12744 \\ 0.439795 & 1. & 0.0572129 & 0.0807283 & 0.180308 \\ -0.0030219 & 0.0572129 & 1. & 0.510381 & 0.666898 \\ 0.044081 & 0.0807283 & 0.510381 & 1. & 0.680531 \\ 0.12744 & 0.180308 & 0.666898 & 0.680531 & 1. \end{pmatrix}$$

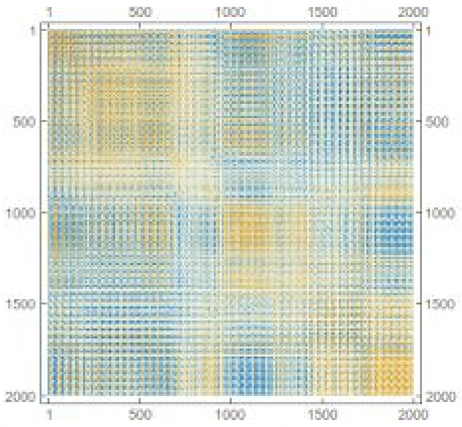
Pixel correlations: attempt 1

This 2000 x2000 matrix shows which pixels are most correlated with eachother, using my first attempted approach. This approach was designed to acknowledge that if two sequences of five white pixels are being compared, their correlation should be high and not subject to image noise. To do this, I applied the normalization steps to each image before transposing them. This produced some *weird* results, including a maximum correlation of 8, which is nonsensical.

```
Clear @ twistedcorrelation
twistedcorrelation[a_] := Module[{o, m, s, b, c},
  o = ConstantArray[{1}, Length @ a];
  m = o.{Mean[a]};
  s = o.{StandardDeviation[a]};
  b = (a - m) / s;
  c = (1 / (Length[a[[1]]] - 1)) * (b.Transpose @ b )(*This multiplication has been reversed intentionally*)
 ]
```

Notice that some of the functions used here (notably correlations[]) are defined in the “imports.m” file I’ve been building throughout this module.

```
MatrixPlot[correlation2 = twistedcorrelation[Transpose @ flattened]]
```



```
Max[Flatten @ correlation2]
```

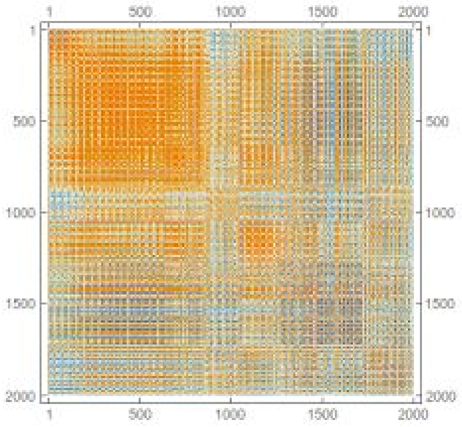
```
8.39199
```

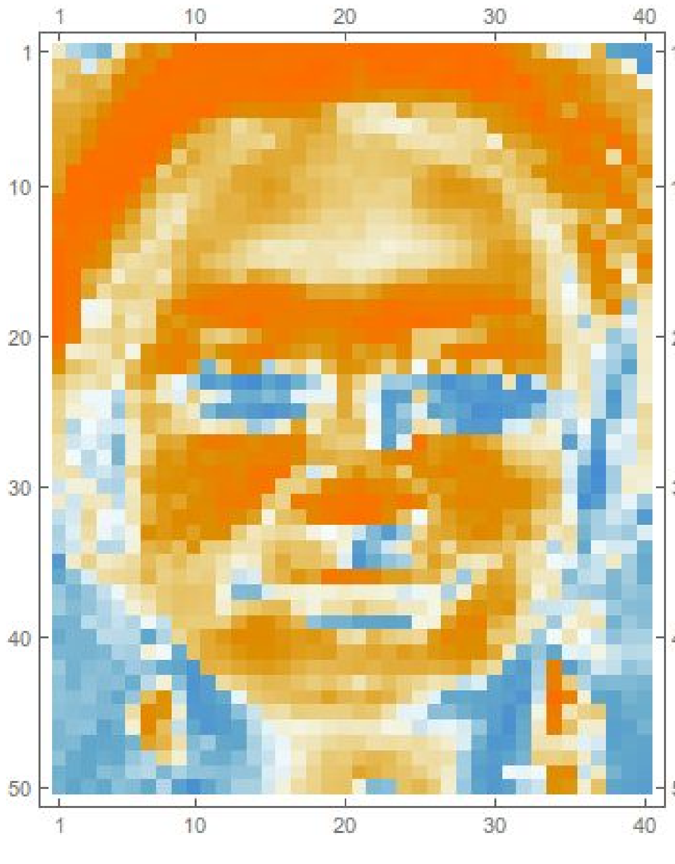
```
correlation2[[1 ;; 10, 1 ;; 10]] // MatrixForm
```

```
( 7.4126   7.75779  7.67101  7.74754  7.25309  4.49013 2.76227 3.29283 1.33189 0.539093 )
( 7.75779  8.17559  8.09546  8.18006  7.62613  4.44571 2.70609 3.16818 1.23749 0.423071 )
( 7.67101  8.09546  8.02129  8.10606  7.55167  4.37071 2.66217 3.11336 1.20662 0.403481 )
( 7.74754  8.18006  8.10606  8.19214  7.62904  4.39638 2.66944 3.12279 1.19664 0.386133 )
( 7.25309  7.62613  7.55167  7.62904  7.12781  4.29022 2.66338 3.13464 1.28005 0.504538 )
( 4.49013  4.44571  4.37071  4.39638  4.29022  4.22456 2.94136 3.79989 1.87743  1.24402 )
( 2.76227  2.70609  2.66217  2.66944  2.66338  2.94136 2.37748 2.86476 1.84798  1.42929 )
( 3.29283  3.16818  3.11336  3.12279  3.13464  3.79989 2.86476 3.69571 2.03312  1.48243 )
( 1.33189  1.23749  1.20662  1.19664  1.28005  1.87743 1.84798 2.03312 1.77301   1.564  )
(0.539093 0.423071 0.403481 0.386133 0.504538 1.24402 1.42929 1.48243  1.564    1.50335 )
```

Pixel correlations: attempt 2

```
MatrixPlot[correlation3 = correlation[flattened]]
```





```
Max[Flatten @ correlation3]
```

```
1.
```

From Data to Dimensions

Vector space

The vector space has dimension 10,000

Magnitudes

$$|A| = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} a_{i,j}^2}$$

Distances

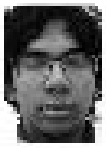
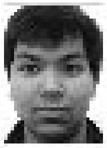
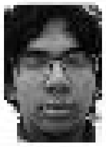
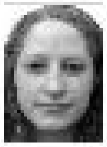
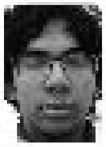
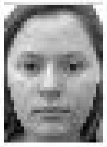
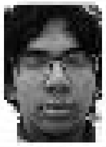
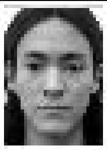
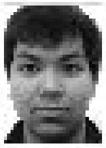
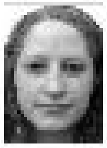
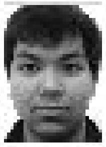
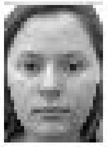
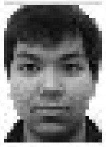
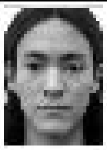
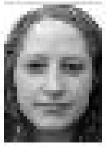
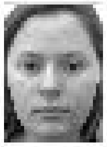
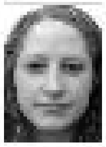
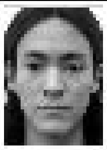
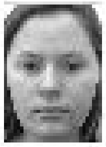
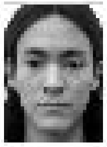
$$|A - B|$$

Dot product

This operation returns a mxm-element matrix showing the (unnormalized) correlations between rows of the input image.

Face distances

I chose to use all five images from before, just for fun.

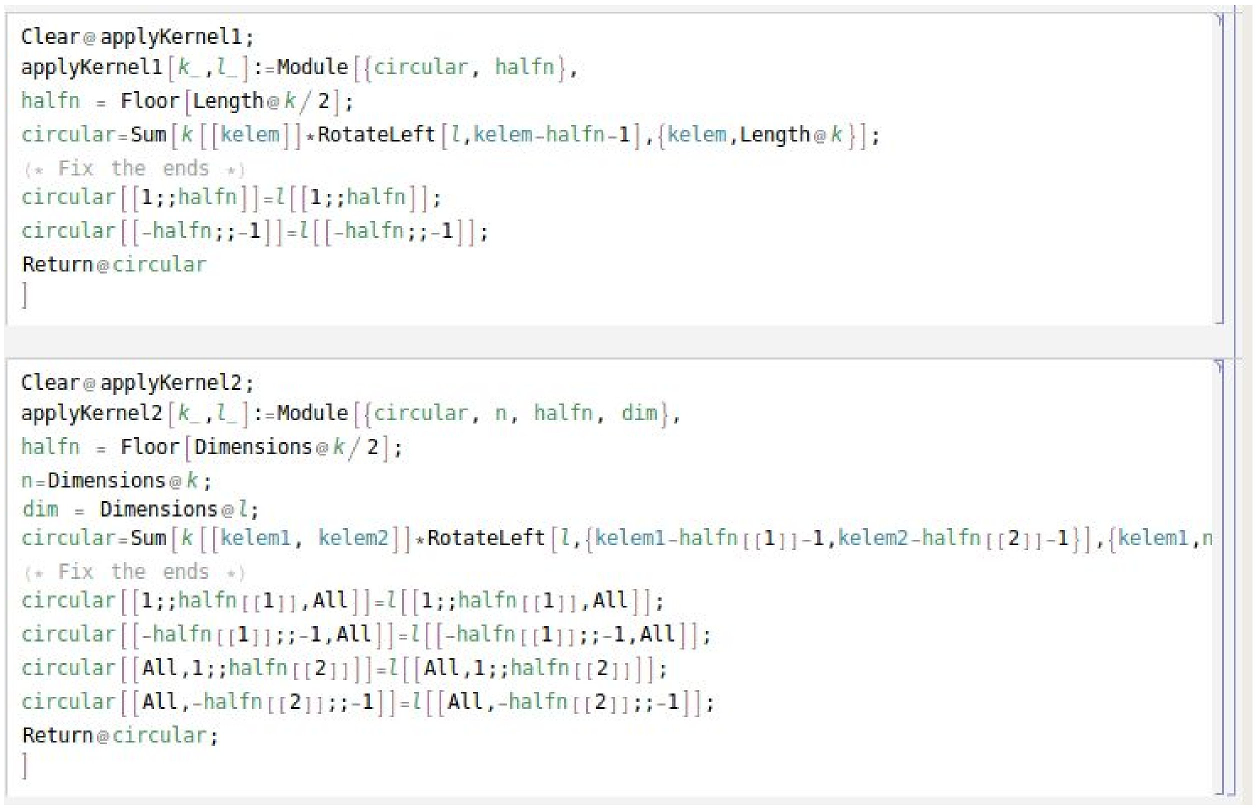
Face 1	Face 2	Distance
		91.5031
		124.921
		121.471
		106.256
		119.855
		113.711
		108.41
		76.8355
		71.6836
		72.5032

Smoothing and Downsampling

Kernel plans

A single twice-nested sum performs much the same operation as for 1D kernels. Discussed in-class.

Kernel implementation



```
Clear@applyKernel1;
applyKernel1[k_,l_]:=Module[{circular, halfn},
halfn = Floor[Length@k/2];
circular=Sum[k[[kelem]]*RotateLeft[l,kelem-halfn-1],{kelem,Length@k}];
(* Fix the ends *)
circular[[1;;halfn]]=l[[1;;halfn]];
circular[[-halfn;;-1]]=l[[-halfn;;-1]];
Return@circular
]
```

```
Clear@applyKernel2;
applyKernel2[k_,l_]:=Module[{circular, n, halfn, dim},
halfn = Floor[Dimensions@k/2];
n=Dimensions@k;
dim = Dimensions@l;
circular=Sum[k[[kelem1, kelem2]]*RotateLeft[l,{kelem1-halfn[[1]]-1,kelem2-halfn[[2]]-1}],{kelem1,n
(* Fix the ends *)
circular[[1;;halfn[[1]],All]]=l[[1;;halfn[[1]],All]];
circular[[-halfn[[1]];;-1,All]]=l[[-halfn[[1]];;-1,All]];
circular[[All,1;;halfn[[2]]]]=l[[All,1;;halfn[[2]]]];
circular[[All,-halfn[[2]];;-1]]=l[[All,-halfn[[2]];;-1]];
Return@circular;
]
```

Tests

applyKernel1 and applyKernel2 come from Imports.m

```
In[121]:= applyKernel1[{0, 2, 0}, Range[10]] // MatrixForm
          applyKernel2[{{0, 0, 0}, {1, 3, 0}, {0, 0, 0}}, IdentityMatrix[5]] // MatrixForm
```

Out[121]//MatrixForm=

$$\begin{pmatrix} 1 \\ 4 \\ 6 \\ 8 \\ 10 \\ 12 \\ 14 \\ 16 \\ 18 \\ 10 \end{pmatrix}$$

Out[122]//MatrixForm=

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 3 & 1 & 0 & 0 \\ 0 & 0 & 3 & 1 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

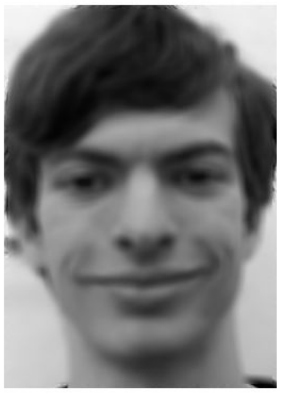
Smoothing

See at right

```
In[171]:= blur = 9;
          kernel = ConstantArray[1 / blur ^ 2, {blur, blur}];
          Image[me]
          Image[applyKernel2[kernel, me]]
```

Out[173]=

Out[174]=

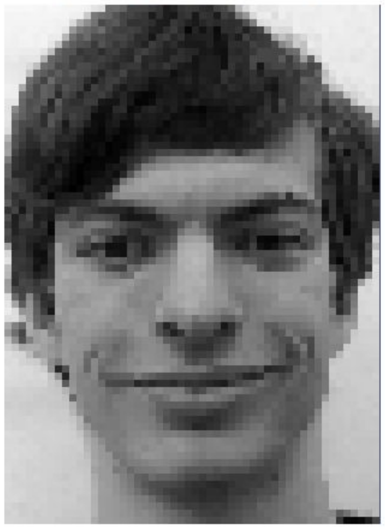


Downsampling

```
In[178]:= downsample[mat_, stride_] := (
            Table[mat[[i, j]], {i, 1, Dimensions[mat][[1]], stride}, {j, 1, Dimensions[mat][[2]], stride}]
           )

In[180]:= Image[downsample[me, 5], ImageSize → Medium]
```

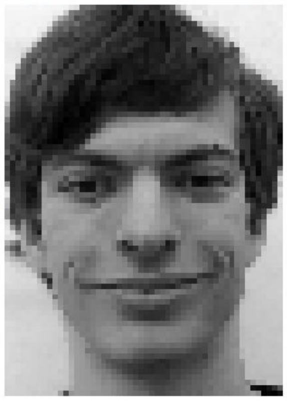
Out[180]=



Builtin downsampling

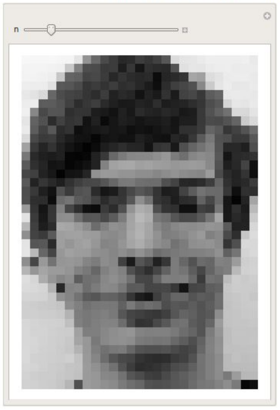
```
In[185]:= Image[ArrayResample[me, Scaled[1 / 5]], ImageSize → Medium]
```

Out[185]=



```
In[196]:= Manipulate[Image[ArrayResample[me, Round @ {1.4 n, n}], ImageSize → Medium], {n, 2, 150}]
```

Out[196]=



Rotation and Scaling

Rotating an image: the plan

I implemented this code in *Mathematica* before even seeing the problem, just trying to match the code from Matlab. My instincts were to try to store all of the data in a single matrix, which would need four rows, as follows:

$$\begin{pmatrix} x \\ y \\ 1 \\ \text{intensity} \end{pmatrix}$$

Ultimately, your approach of storing separate intensities was cleaner.

Rotating an image: implementation

I reimplemented this whole thing, and it's kind of nice. Code below.

```
In[10]:= Clear @ unconvert
         unconvert[tranxypos_, intensities_] := Module[{m, n, xytran, imageData},
           {m, n} = ImageDimensions[picture];
         xytran = Transpose[Ceiling[tranxypos]];
           xytran[[All, 3]] = intensities;
         xytran = Cases[xytran, {x_, y_, _} /; (1 ≤ x ≤ m && 1 ≤ y ≤ n)];
           imageData = ConstantArray[0, {n, m}];
         Table[
            imageData[[i[[2]], i[[1]]]] = i[[3]];
            , {i, xytran}];
           imageData
          ]
```

Because images are lists of rows, this intensities list traverses one row at a time.

```
In[12]:= Clear[convert];
         convert[picture] := Module[{m, n, intensities, xydata},
           {m, n} = ImageDimensions[picture];
           (* intensities is put into a global variable, which isn't optimal, but works *)
           intensities = Flatten[ImageData[ColorConvert[picture, "Grayscale"]]];
         (* m is width, n is height *)
           xydata = Transpose @ Flatten[Table[{x, y, 1}, {y, 1, n}, {x, 1, m}], 1];
           {xydata, intensities}
          ]
```

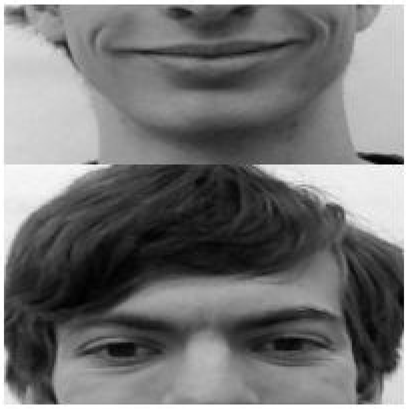
Other image processing [done?]

Shifting an image

```
In[33]:= translator = DiagonalMatrix[ConstantArray[1, 199], -1];
         translator[[1, 200]] = 1;

In[35]:= Image[MatrixPower[translator, 80].square]
```

Out[35]=

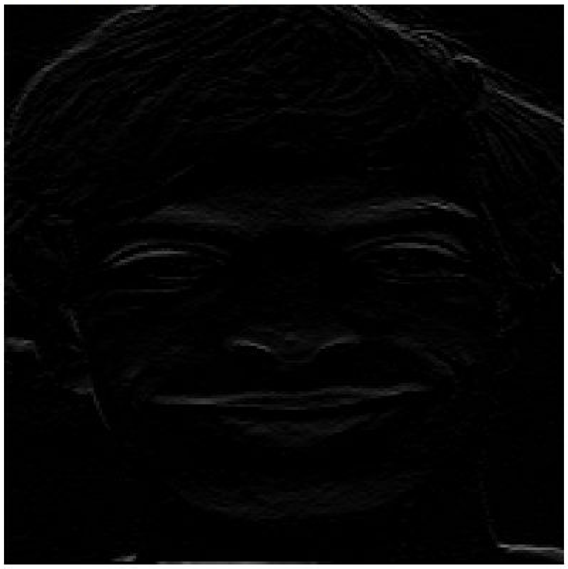


Edge detection

```
In[36]:= edgedetect = DiagonalMatrix[ConstantArray[1, 200]] + DiagonalMatrix[ConstantArray[-1, 199], 1];
         edgedetect[[200, 1]] = -1;

In[40]:= Image[edgedetect.square]
```

Out[40]=

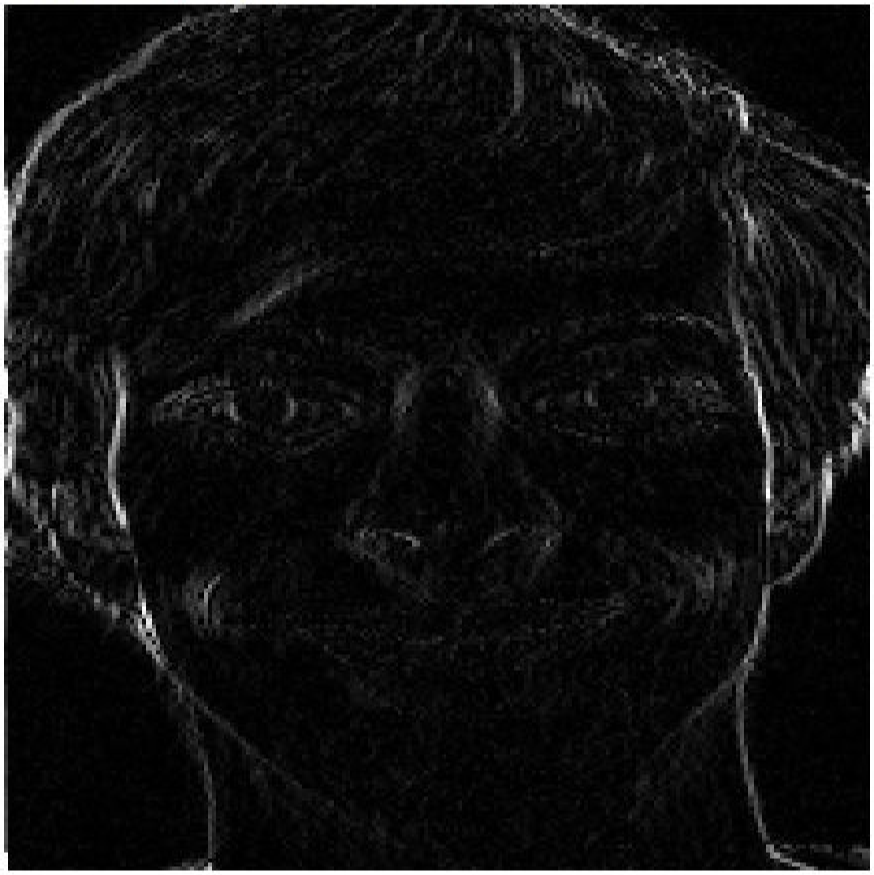


Directionality

This multiplication only detects horizontal edges. This results from the fact that each row of the transformation matrix is mapped to a column of the image, and detects edges in that column vector. If the multiplication were done the other way, it would detect horizontal edges.

```
In[44]:= Image[3 * Abs[square.edgedetect]]
```

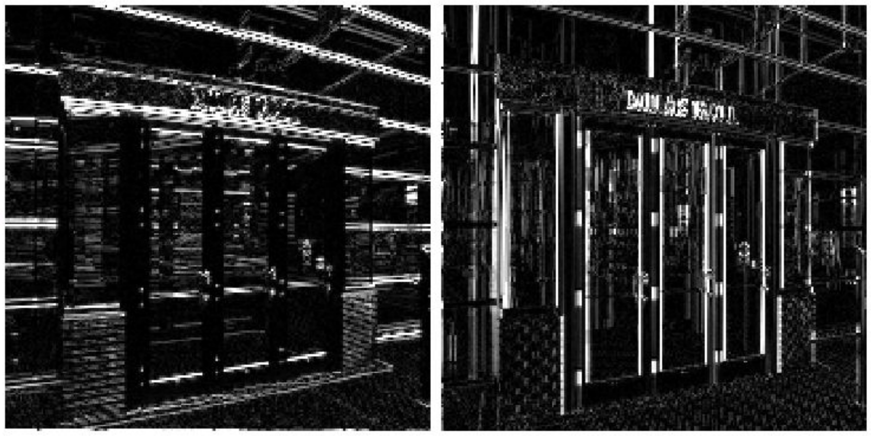
Out[44]=



Samples

```
In[50]:= Grid[{{Image[3 * Abs[edgedetect.milas], ImageSize → Medium], Image[3 * Abs[milas.edgedetect], ImageSize → Medium]}}]
```

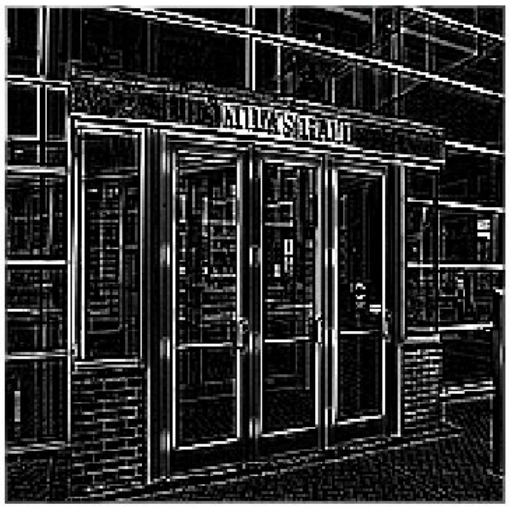
Out[50]=



Sample kernel 1

```
In[52]:= Image @ applyKernel2[( -1 -1 -1
                                -1  8 -1
                                -1 -1 -1 ), milas]
```

Out[52]=



Sample kernel 2

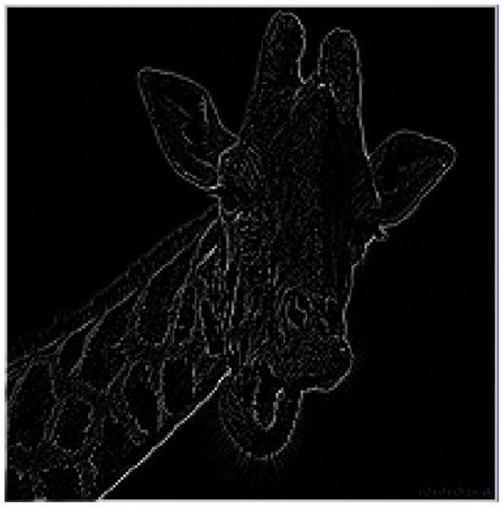
As expected, these are nice, direction-independent, edge detectors. They do have a bias towards the orthogonal axes of the image, especially the second.

```
In[53]:= giraffe = ImageData[ColorConvert[ImageResize[
```

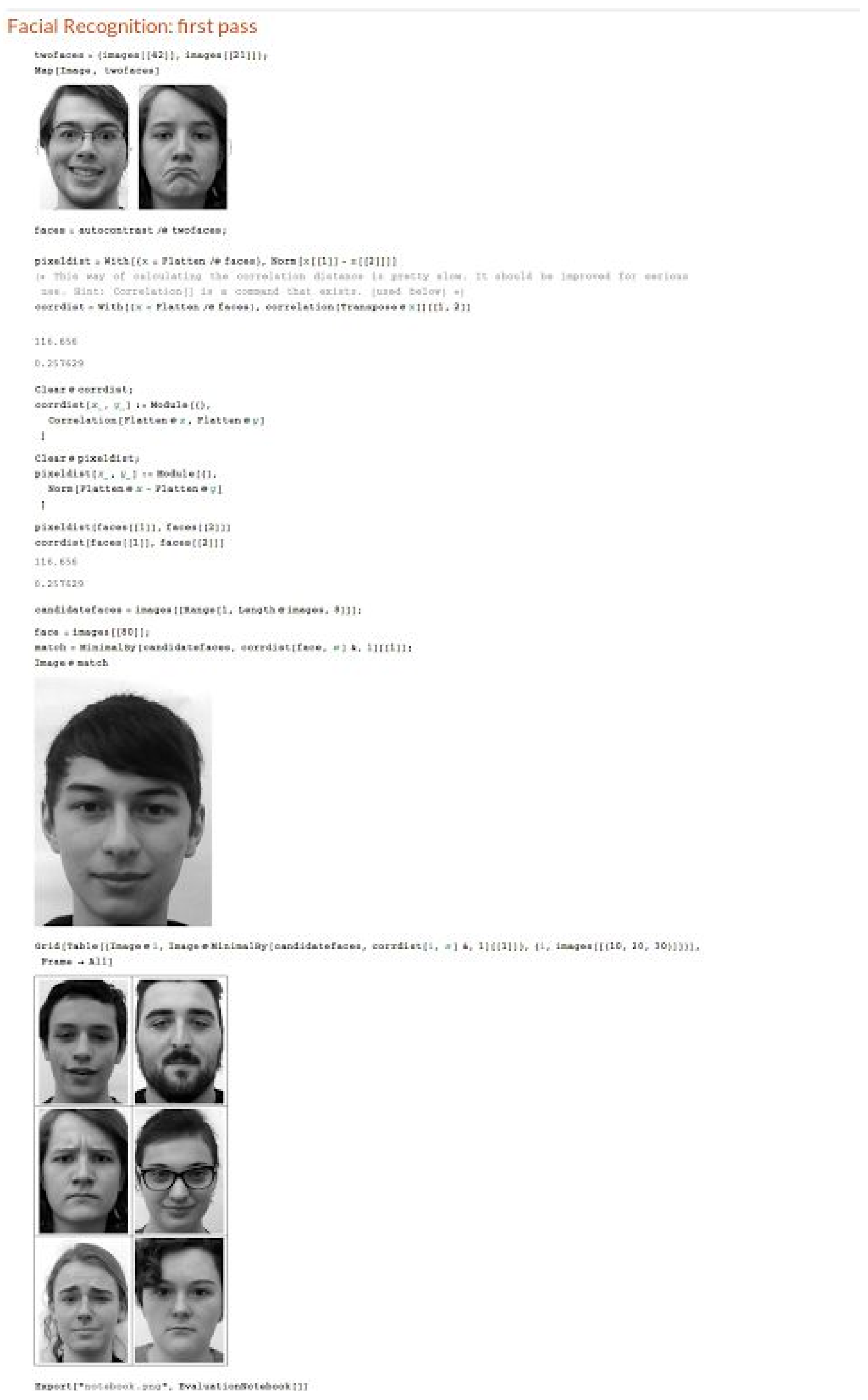
```
, {200, 200

In[54]:= Image @ applyKernel2[(  0 -1  0
                                -1  4 -1
                                 0 -1  0 ), giraffe]
```

Out[54]=



A first attempt at facial recognition



Facial Recognition: first pass

```
twofaces = {images[[42]], images[[21]]};
Map[Image, twofaces]
```

```
faces = autocontrast /@ twofaces;

pixeldist = With[{x = Flatten /@ faces}, Norm[x[[1]] - x[[2]]]]
(* This way of calculating the correlation distance is pretty slow. It should be improved for serious
 use. Hint: Correlation[] is a command that exists. (used below) *)
corrdist = With[{x = Flatten /@ faces}, correlation[Transpose @ x]][[1, 2]]
```

```
116.656

0.257629
```

```
Clear @ corrdist;
corrdist[x_, y_] := Module[{},
  Correlation[Flatten @ x, Flatten @ y]
 ]

Clear @ pixeldist;
pixeldist[x_, y_] := Module[{},
  Norm[Flatten @ x - Flatten @ y]
 ]

pixeldist[faces[[1]], faces[[2]]]
corrdist[faces[[1]], faces[[2]]]
```

```
116.656

0.257629
```

```
candidatefaces = images[[Range[1, Length @ images, 8]]];

face = images[[80]];
match = MinimalBy[candidatefaces, corrdist[face, #] &, 1][[1]];
Image @ match
```

```
Grid[Table[{Image @ i, Image @ MinimalBy[candidatefaces, corrdist[i, #] &, 1][[1]]}, {i, images[[{10, 20, 30}]]}],
 Frame → All]
```

```
Export["notebook.png", EvaluationNotebook[]]
```