IB Paper 8: Photo Editing

Lecture 2: Resizing and Rotating

N G Kingsbury (given by J Lasenby in 2016)

Signal Processing Group,
Engineering Department,
Cambridge, UK

Easter 2016
Resizing the image

- This section is concerned with **resizing** the image so that we have fewer or more pixels.

- Often want to do this: to save storage; to combine images; to do operations on multiple images etc.

- First look at the issues (**aliasing etc – See SDA IB**) involved in resampling/resizing an image. **Very** important in most image processing considerations.

- Then look at the script `ph_resize`

- Then look at `im_resize` which is the function that performs the interpolation

- Finally look at the actual processes of **bi-linear** and **bi-cubic** interpolation.
Example of Aliasing - I
Illustration of spatial frequencies in images

Figure 1: 1d sine wave

Figure 2: Spectrum of the 1d sine wave

Figure 3: 1d sine wave – varying frequency

Figure 4: Spectrum of variable frequency sine wave
Figure 5: 2d sine wave

Figure 6: Spectrum of the 2d sine wave

Figure 7: 2d sine wave – varying frequency [spatial tilt]

Figure 8: Spectrum of variable frequency 2d sine wave
Figure 9: 2d texture

Figure 10: Spectrum of the texture

Figure 11: 2d texture – varying frequency [spatial tilt]

Figure 12: Spectrum of variable frequency 2d texture
The basic script: ph_resize

- As for **ph_crop**, **ph_resize** contains 5 cases which are selected by **mode**.
- **init**: performs initialisation and sets up the command box to enter parameters
- **Set size**: called when the **X size** and/or **Y size** boxes are modified. **Scale** is then blanked (for obvious reasons).
- **Set scale**: called when **Scale** box is modified.
- **Resize now**: called when the **Resize now** button is pressed – this calls **im_resize** and then **showimages**.
- **Close**: closes command box and redisplays the **Before** and **After** images.
The function `imresize`

- Vectors \texttt{sx} and \texttt{sy} give sizes of the input and output images.
- \texttt{ri} and \texttt{ci} contain the sampling points (row and column) of the output image: eg if input image is 4 pixels wide, ie with samples at [0.5 1.5 2.5 3.5], doubling the size of the image means the new output columns, \texttt{ci}, will be

\[
\text{ci} = [0.25 0.75 1.25 1.75 2.25 2.75 3.25 3.75]
\]

- To halve the size of the image, the pixels would be 2 units wide and \texttt{ci} would be [1.0 3.0]
- ie if \texttt{sizeout} and \texttt{sizein} are the sizes of the output and input images, we create these arrays as follows

\[
\text{ci} = [0.5 : \text{sizeout} - 0.5] \times (\text{sizein})/(\text{sizeout})
\]

- the Matlab function `interp2` is used to interpolate our input image \texttt{xui} to produce an output image \texttt{yui}. 
Input array vs Output array

On the previous slide we had:

if `sizeout` and `sizein` are the sizes of the output and input images, we create these arrays as follows

\[ ci = [0.5 : sizeout - 0.5] \ast (sizein)/(sizeout) \]

The Matlab syntax \([0.5 : sizeout - 0.5]\) gives an array increasing in steps of 1.

ie if `sizeout = 8`, then

\[ ci = [0.5 : 7.5] = [0.5, 1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5] \]
Linear interpolation – 1D

In 1-D, if we have pixels $x_a$ and $x_b$, sampled at locations $a$ and $b$, then the linearly interpolated pixel at location $p$ (assumed to lie somewhere between $a$ and $b$) is given by:

$$x_p = x_a + \frac{p - a}{b - a} (x_b - x_a) = \frac{(b - p)x_a + (p - a)x_b}{b - a} \quad (1)$$

A simple example of this is shown below.
Bi-linear interpolation – 2D

In 2-D, the equivalent operation is bi-linear interpolation, in which the interpolated pixel at \( \{p, q\} \) is given in terms of the four surrounding pixels

\[
\begin{align*}
\mathbf{x}_{pq} &= \frac{(d - q) \left[ (b - p)x_{a,c} + (p - a)x_{b,c} \right] + (q - c) \left[ (b - p)x_{a,d} + (p - a)x_{b,d} \right]}{(b - a)(d - c)}
\end{align*}
\]
Bi-linear interpolation – 2D

This is found by performing 1-D linear interpolation twice, first down the columns and then across the rows (or vice-versa).

\[
x(p, c) = x(a, c) + \frac{p - a}{b - a} (x(b, c) - x(a, c))
\]

\[
x(p, d) = x(a, d) + \frac{p - a}{b - a} (x(b, d) - x(a, d))
\]

\[
x(p, q) = x(p, c) + \frac{q - c}{d - c} (x(p, d) - x(p, c))
\]

Done for each of the colour channels.
Cubic interpolation - 1D

In 1-D, cubic interpolation to a point \( p \) uses 4 consecutive samples (instead of 2) located at \( \{a, b, c, d\} \), such that \( b \leq p \leq c \), to produce an interpolated point

\[
x_p = \frac{1}{2} \left[ -u(1-u)^2 x_a + (1-u)(2+2u-3u^2)x_b + u(1+4u-3u^2)x_c - u^2(1-u)x_d \right]
\]

where \( u = \frac{p-b}{c-b} \), \( u \) goes from 0 to 1

so that \( u = 0 \) when \( p \) is at \( b \), and \( u = 1 \) when \( p \) is at \( c \). The sampling points \( \{a, b, c, d\} \) should be uniformly spaced. The above expression is found by solving for \( \alpha, \beta, \gamma, \delta \) in the following

\[
x(u) = \alpha u^3 + \beta u^2 + \gamma u + \delta
\]

with constraints that \( x(0) = x_b, x(1) = x_c \) and \( \frac{dx}{du} = \frac{x_c-x_a}{2} \) at \( u = 0 \) and \( \frac{dx}{du} = \frac{x_d-x_b}{2} \) at \( u = 1 \).
Interpolation ensures continuity of gradient and value, as well as giving a high degree of smoothness when the input points lie on a straight line.
This is the Fourier transform/frequency response of an impulse response (first 6 samples in previous figure) for the two forms of interpolation. Note: cubic gives greater flatness (lower sidelobes) at high frequencies and greater gain in the mainlobe.
In 2-D we apply the 1-D equation in both vertical and horizontal directions to the $4 \times 4$ neighbourhood of pixels surrounding $x_{p,q}$. In `interp2` we select the argument `cubic`. The expression for $x_{p,q}$ is fairly messy!
Rotating the image: \texttt{ph\_rotate}

\texttt{ph\_rotate} is concerned with rotating the image – either to correct for gross rotations of the camera (e.g. through \( \pm 90^\circ \)) or for small errors which result in sloping horizons or non-true verticals etc. The function \texttt{im\_rotate} does this via use (again) of the \texttt{interp2} Matlab function.

- Comprises 5 cases selected by \texttt{mode} (as before) – but note that one case \texttt{Rotate} uses a separate \texttt{switch} statement.

- \texttt{Init}: sets up control window
- \texttt{Slider}: reads the slider value and calls \texttt{Rotate}
- \texttt{Edit Box}: sets angle to value in the edit box and calls \texttt{Rotate}
- \texttt{Close}: closes and redisplay before and after images.
- \texttt{Rotate}: rotates via a call to \texttt{im\_rotate()}. 
The function: \texttt{imrotate()}

- First checks if the rotation is simply a multiple of 90°.
- If rotation angle is $\theta$, form the rotation matrix
  \[
  R = \begin{bmatrix}
  \cos \theta & \sin \theta \\
  -\sin \theta & \cos \theta
  \end{bmatrix}
  \]
- Then work out size of output image necessary to contain the rotated image
- If $p$ is rotated vector $u$ (in original coordinates) then we have $p = R^T u$ from which we have the formula in the notes $[u, v] = [p, q] R^T$.
- Then need to add on an offset (according to where we rotate about) to ensure new image coordinates are measured from top LHC.
- Call \texttt{interp2()} since $[u, v]$ above will not necessarily be at pixel locations – use bi-linear interpolation.
Rotating the Coordinate vectors

For clarification:

\[ R^T = \begin{bmatrix}
\cos \theta & -\sin \theta \\
\sin \theta & \cos \theta
\end{bmatrix} \]

is the matrix which rotates *anticlockwise* through an angle \( \theta \). Therefore, rotating \( u = [u, v]^T \) through \( \theta \) gives a new vector \( p = [p, q]^T \):

\[ p = R^T u \]

Taking the transpose of this gives \( p^T = u^T R \). Multiply both sides by \( R^T \) on the right, to give

\[ p^T R^T = u^T \]

Thus putting everything in terms of *row vectors*.
Summary

• Section 3 of the notes outlines how the Photo Editor resizes images using various types of interpolation.
• Section 4 deals with rotating the image.
• Both rely crucially on the interp2() interpolation function in Matlab.

J. Lasenby (Easter 2016)