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

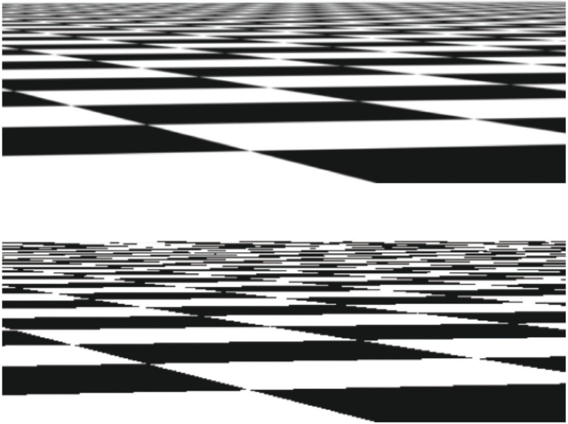


Illustration of spatial frequencies in images

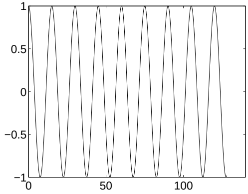


Figure 1: 1d sine wave

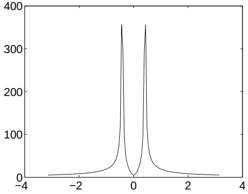


Figure 2: Spectrum of the 1d sine wave

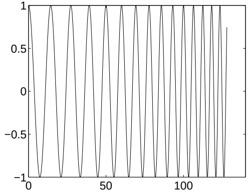


Figure 3: 1d sine wave – varying frequency

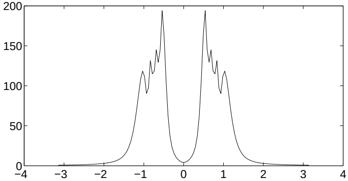


Figure 4: Spectrum of variable frequency sine wave

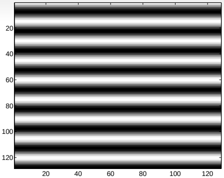


Figure 5: 2d sine wave

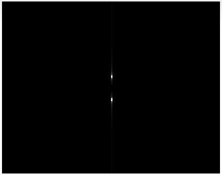


Figure 6: Spectrum of the 2d sine wave

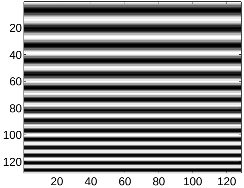


Figure 7: 2d sine wave – varying frequency [spatial tilt]

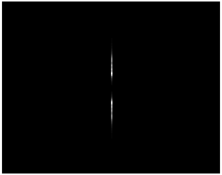


Figure 8: Spectrum of variable frequency 2d sine wave

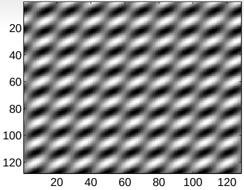


Figure 9: 2d texture

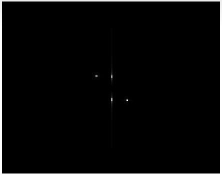


Figure 10: Spectrum of the texture

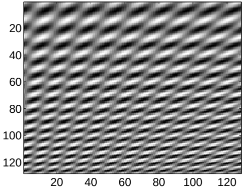


Figure 11: 2d texture – varying frequency [spatial tilt]

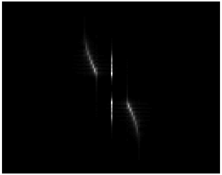


Figure 12: Spectrum of variable frequency 2d texture

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 `im_resize`

- Vectors `sx` and `sy` give sizes of the input and output images.
- `ri` and `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, `ci`, will be

$$ci = [0.25\ 0.75\ 1.25\ 1.75\ 2.25\ 2.75\ 3.25\ 3.75]$$

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

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

- the Matlab function `interp2` is used to interpolate our input image *xui* to produce an output image *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

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

The Matlab syntax $[0.5 : \text{sizeout} - 0.5]$ gives an array increasing in steps of 1.
ie if $\text{sizeout} = 8$, then

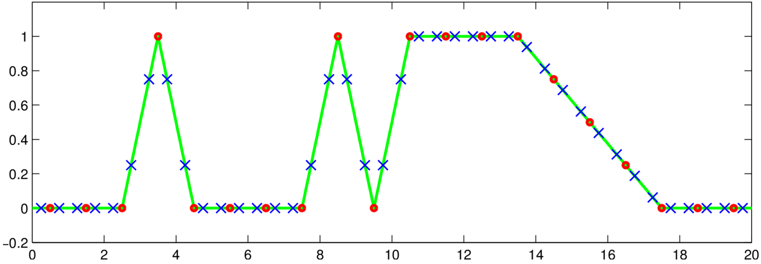
$$ci = [0.5 : 7.5] = [0.5, 1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5]$$

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:

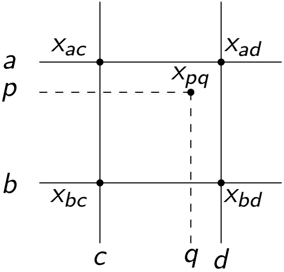
$$x_p = x_a + \frac{p-a}{b-a}(x_b - x_a) = \frac{(b-p)x_a + (p-a)x_b}{b-a} \qquad (1)$$

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

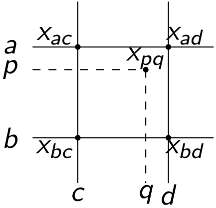


by

$$x_{p,q} = \frac{(d-q)\ [(b-p)x_{a,c} + (p-a)x_{b,c}] + (q-c)\ [(b-p)x_{a,d} + (p-a)x_{b,d}]}{(b-a)(d-c)}$$

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

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

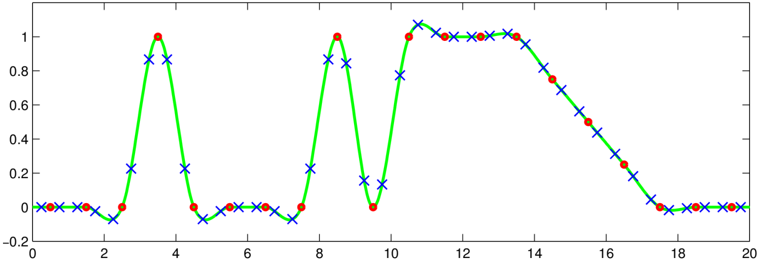
$$\text{where} \qquad u = \frac{p-b}{c-b}, \qquad u \text{ goes from } 0 \text{ to } 1$$

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 α,β,γ,δ in the following

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

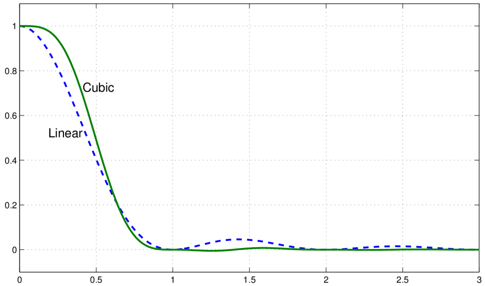
with constraints that $x(0) = x_b$, $x(1) = x_c$ and $\frac{dx}{du} = \frac{x_c - x_a}{2}$ at $u = 0$ and $\frac{dx}{du} = \frac{x_d - x_b}{2}$ at $u = 1$.

Cubic interpolation - 1D: Example



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.

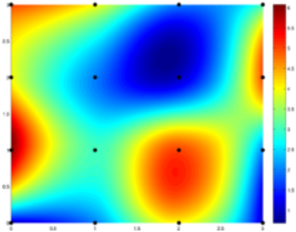
Frequency Response – linear vs cubic



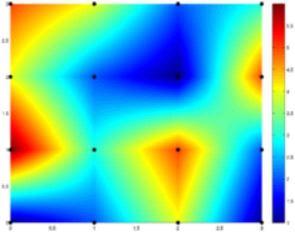
frequency / (sampling frequency of input data)

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.

Bi-cubic interpolation - 2D



Bi-cubic interpolation



Bi-linear interpolation

In 2-D we apply the 1-D equation in both vertical and horizontal directions to the 4×4 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: `ph_rotate`

`ph_rotate` is concerned with rotating the image – either to correct for gross rotations of the camera (e.g. through ±90°) or for small errors which result in sloping horizons or non-true verticals etc. The function `im_rotate` does this via use (again) of the `interp2` Matlab function.

- Comprises 5 cases selected by **mode** (as before) – but note that one case **Rotate** uses a separate **switch** statement.
- **Init**: sets up control window
- **Slider**: reads the slider value and calls **Rotate**
- **Edit Box**: sets angle to value in the edit box and calls **Rotate**
- **Close**: closes and redisplays before and after images.
- **Rotate**: rotates via a call to `im_rotate()`.

The function: `im_rotate()`

- First checks if the rotation is simply a multiple of 90°.
- If rotation angle is θ, 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 $\mathbf{p} = R^T\mathbf{u}$ 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 `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 θ. Therefore, rotating $\mathbf{u} = [u, v]^T$ through θ gives a new vector $\mathbf{p} = [p, q]^T$:

$$\mathbf{p} = R^T\mathbf{u}$$

Taking the transpose of this gives $\mathbf{p}^T = \mathbf{u}^T R$. Multiply both sides by R^T on the right, to give

$$\mathbf{p}^T R^T = \mathbf{u}^{\mathbf{T}}$$

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)