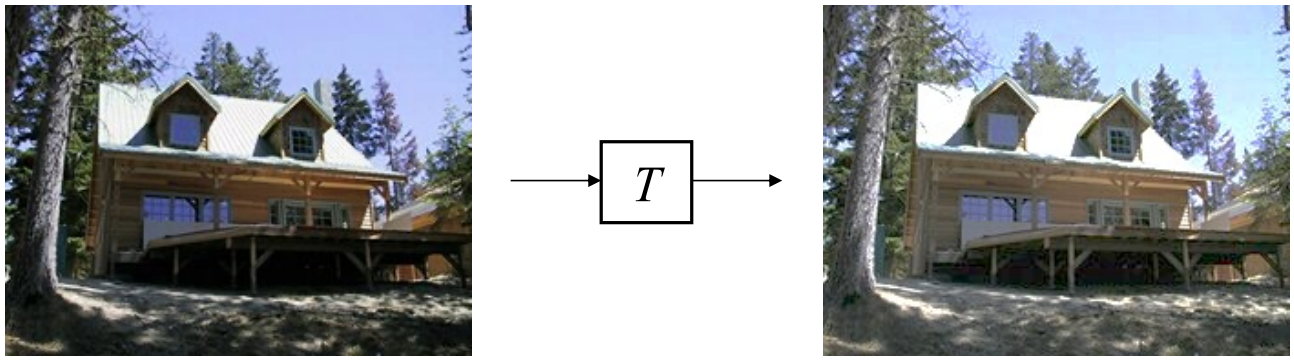
Image transformations

Contrast change



Point processing

- Change *range* of image:
  - Apply the same function to each pixel value

- Many applications

- Common
  - Linear sensors produce strange pictures
    - The world has high dynamic range, 8 bits is too few
    - Typically 12 bit sensor, 8 bit image
  - Most cameras process sensor results before they report them

From wikipedia

From wikipedia

From wikipedia

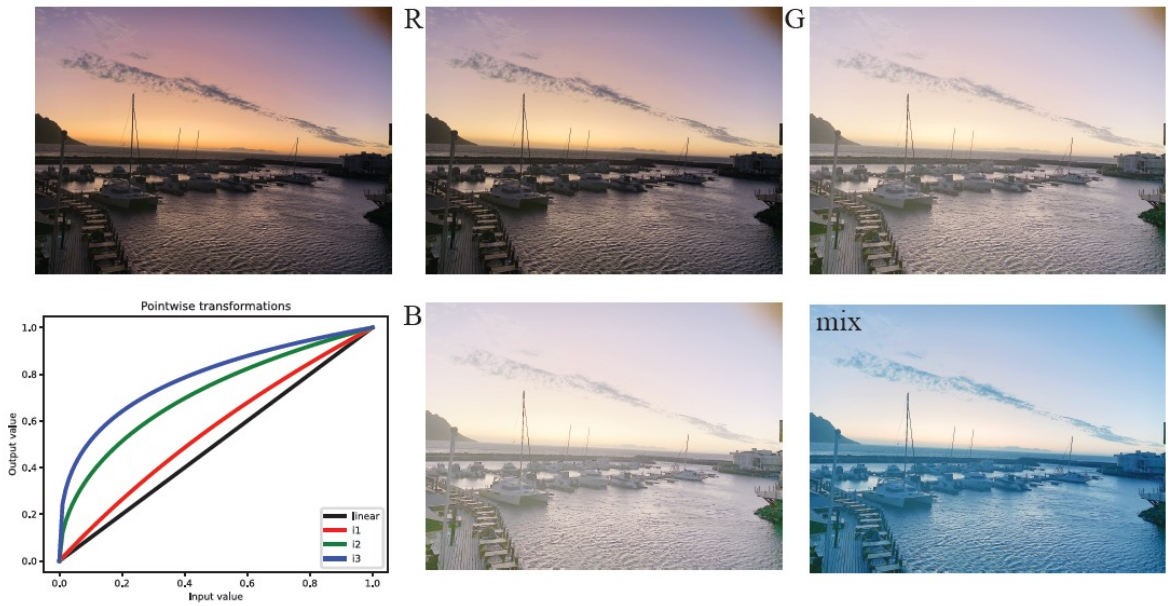
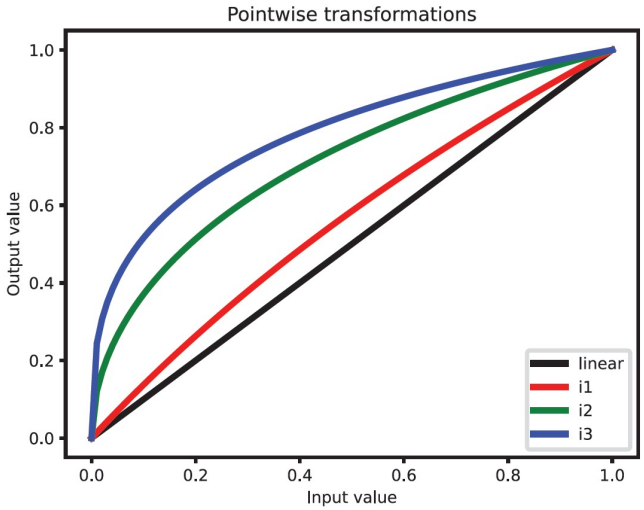


FIGURE 3.1: *A number of pointwise image transformations applied to the image on the* **top left**. *The* **bottom left** *shows plots of the function applied; on the* **right**, *results of applying these functions to that image. These transformations tend to spread out the dark values, and squash the brighter values.* **R**, **G** *and* **B** *respectively show the red, green and blue functions applied to each of the color channels of the image.* **Mix** *shows the result of applying the red function to the red channel, the green function to the green channel, and the blue function to the blue channel. Darker pixels tend to shift to the blue in the mix result, and brighter pixels have a less pronounced color shift.* Image credit: *Figure shows my photograph of a sunset at Gordon's Bay.*

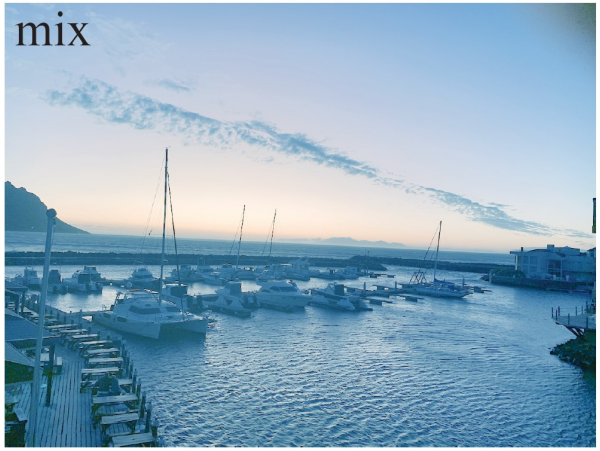
R

G


Pointwise transformations
Output value
Input value
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
linear
i1
i2
i3

B

mix



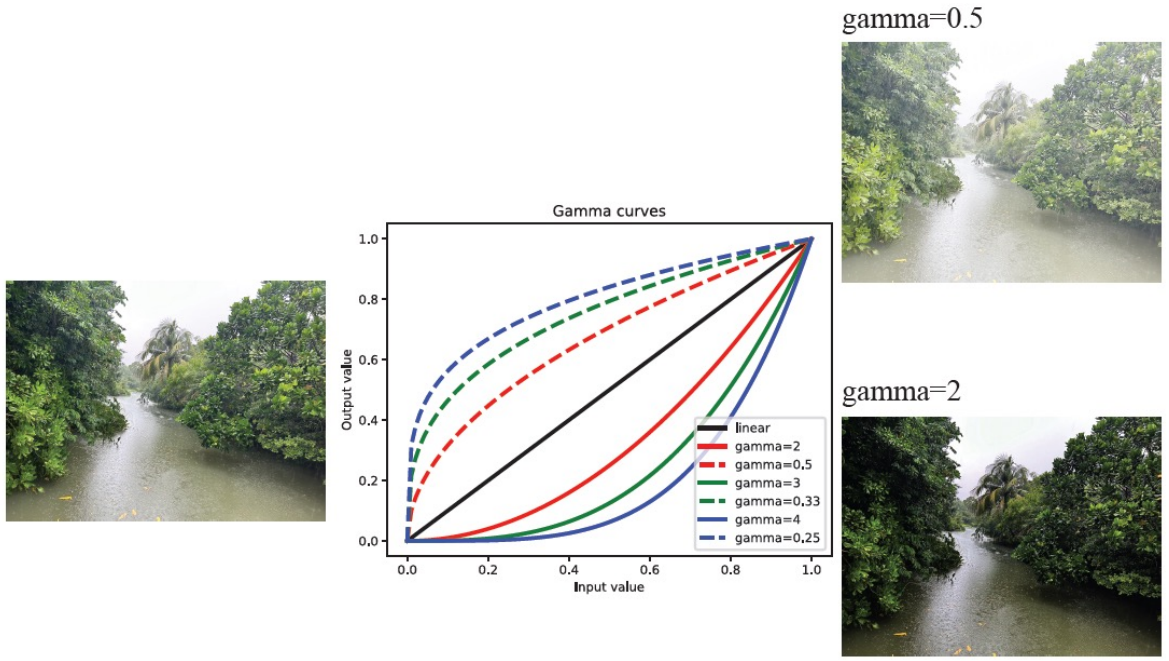
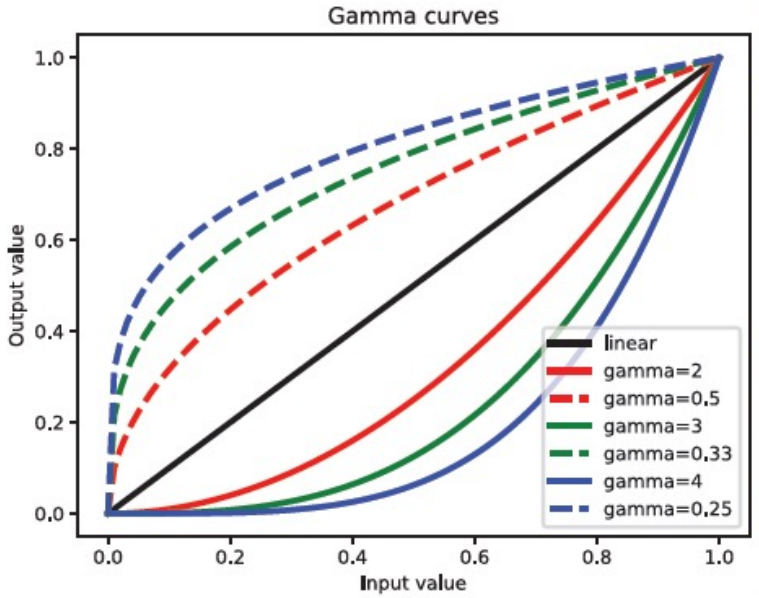


FIGURE 3.2: *Many imaging and rendering devices have a response that is a power of the input, so that output* $= C input^{\gamma}$*, where* γ *is a parameter of the device. One can simulate this effect by applying a transform like those shown in the* **center** *(curves for several values of* γ*). Note that you can remove the effect of such a transform* – gamma correct *the image – by applying another such transform with an appropriately chosen* γ*. The image on the* **left** *is transformed to the two examples on the* **right** *with different* γ *values.* Image credit: *Figure shows my photograph of a river in Singapore.*



gamma=0.5

gamma=2

Transformations

- Notation:

that transformations are invertible. Adopt the convention that a point $\mathbf{x} = (x, y)$ is mapped by a transformation to the point $\mathbf{u} = (u, v) = (u(x, y), v(x, y))$, and $\mathbf{u} = (u, v)$ is mapped to $\mathbf{x} = (x, y)$ by the inverse. In vector notation, $\mathbf{x}$ is mapped to $\mathbf{u}$, and so on. Write $\mathbf{A}$ for a 2×2 matrix, whose i, j'th component is a_{ij}.

Translation and rotation

Translation maps the point (x, y) to the point $(u, v) = (u(x, y), v(x, y)) = (x + t_x, y + t_y)$ for two constants t_x and t_y. Here $(x, y) = (u - t_x, v - t_y)$. In vector notation, $\mathbf{u} = \mathbf{x}+\mathbf{t}$ and $\mathbf{x} = \mathbf{u}-\mathbf{t}$. Translation preserves lengths and angles. Choose two points $\mathbf{x}_1$ and $\mathbf{x}_2$. The squared distance from $\mathbf{x}_1$ to $\mathbf{x}_2$ is $(\mathbf{x}_1 - \mathbf{x}_2)^T(\mathbf{x}_1 - \mathbf{x}_2)$; but for a translation $(\mathbf{u}_1 - \mathbf{u}_2) = (\mathbf{x}_1 - \mathbf{x}_2)$. A similar argument shows that angles are preserved (**exercises**).

Rotation takes the point (x, y) to the point $(u, v) = (u(x, y), v(x, y)) = x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta$. Here θ is the angle of rotation, rotation is anti-clockwise, and $(x, y) = u\cos\theta + v\sin\theta, -u\sin\theta + v\cos\theta$. Write $\mathcal{R}$ for a rotation matrix (a matrix where $\mathcal{R}^T\mathcal{R} = \mathcal{I}$ and $\det(\mathcal{R}) = 1$); then $\mathbf{u} = \mathcal{R}\mathbf{x}$ and $\mathbf{x} = \mathcal{R}^{-1}\mathbf{u} = \mathcal{R}^T\mathbf{u}$. Rotation preserves lengths and angles. Choose two points $\mathbf{x}_1$ and $\mathbf{x}_2$. The squared distance from $\mathbf{x}_1$ to $\mathbf{x}_2$ is $(\mathbf{x}_1 - \mathbf{x}_2)^T(\mathbf{x}_1 - \mathbf{x}_2)$; but for a rotation $(\mathbf{u}_1 - \mathbf{u}_2) = \mathcal{R}(\mathbf{x}_1 - \mathbf{x}_2)$ and $\mathcal{R}^T\mathcal{R} = \mathcal{I}$. A similar argument shows that angles are preserved (**exercises**).

Euclidean Transformations

A Euclidean transformation is a rotation and translation, so $(u(x,y), v(x,y)) = (x\cos\theta - y\sin\theta + t_x, v(x,y) = x\sin\theta + y\cos\theta + t_y)$. Euclidean transformations preserve lengths and angles (and so areas) and are sometimes referred to as rigid body transformations. Here $(x,y) = ((u-t_x)\cos\theta + (v-t_y)\sin\theta, -(u-t_x)\sin\theta + (v-t_y)\cos\theta)$. In vector notation, $\mathbf{u} = \mathcal{R}\mathbf{x} + \mathbf{t}$ and $\mathbf{x} = \mathcal{R}^{-1}(\mathbf{u}-\mathbf{t}) = \mathcal{R}^T(\mathbf{u}-\mathbf{t})$. Euclidean transformations preserve lengths and angles (you can think of a Euclidean transformation as a rotation followed by a translation).

Scaling

Uniform scaling where $(u, v) = (sx, sy)$ for $s > 0$. Here $(x, y) = (1/su, 1/sv)$. In vector notation, $\mathbf{u} = s\mathbf{x}$ and $\mathbf{x} = (1/s)\mathbf{u}$. Uniform scaling preserves angles, but not lengths (**exercises**).

Non-uniform scaling where $(u, v) = (sx, ty)$ for s and t both positive, and so $(x, y) = (1/su, 1/tv)$. Write $\text{diag}(s, t)$ for the matrix with s and t on the diagonal. In vector notation, $\mathbf{u} = \text{diag}(s, t)\mathbf{x}$ and $\mathbf{x} = \text{diag}(1/s, 1/t)\mathbf{u}$. Non-uniform scaling will usually change both lengths and angles.

Affine transformations

Affine transformations are better written in vector notation. Write $\mathcal{A}$ for a 2×2 matrix which is invertible, and $\mathbf{t}$ for some constant vector. Here $\mathbf{u} = \mathcal{A}\mathbf{x} + \mathbf{t}$ and $\mathbf{x} = \mathcal{A}^{-1}(\mathbf{u} - \mathbf{t})$. Affine transformations will usually change both lengths and angles.

Proj

Projective transformations involve quite inefficient notation if one does not know homogenous coordinates (Section **??**), and writing them in vector form is clumsy. Write p_{ij} for the i, j'th component of a 3×3 matrix $\mathcal{P}$ that is invertible. Then

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \frac{p_{11}x+p_{12}y+p_{13}}{p_{31}x+p_{32}y+p_{33}} \\ \frac{p_{21}x+p_{22}y+p_{23}}{p_{31}x+p_{32}y+p_{33}} \end{bmatrix}.$$

The inverse transformation is obtained by applying the inverse of $\mathcal{P}$ to $\mathbf{u}$ according to the recipe above. Notice that all the classes of transformation described are a case of a projective transformation (**exercises**). For a vector representation, write

$$\mathcal{P} = \begin{bmatrix} \mathbf{p}_1^T & p_{13} \\ \mathbf{p}_2^T & p_{23} \\ \mathbf{p}_3^T & p_{33} \end{bmatrix}$$

for a 3×3 array with inverse $\mathcal{Q}$. Then

$$\mathbf{u} = \begin{bmatrix} \frac{\mathbf{p}_1^T\mathbf{x}+p_{13}}{\mathbf{p}_3^T\mathbf{x}+p_{33}} \\ \frac{\mathbf{p}_2^T\mathbf{x}+p_{23}}{\mathbf{p}_3^T\mathbf{x}+p_{33}} \end{bmatrix} \text{ and } \mathbf{x} = \begin{bmatrix} \frac{\mathbf{q}_1^T\mathbf{u}+q_{13}}{\mathbf{q}_3^T\mathbf{u}+q_{33}} \\ \frac{\mathbf{q}_2^T\mathbf{u}+q_{23}}{\mathbf{q}_3^T\mathbf{u}+q_{33}} \end{bmatrix}$$

Notice also that if $\mathcal{P} = \lambda\mathcal{Q}$ for some $\lambda \neq 0$, then $\mathcal{P}$ and $\mathcal{Q}$ implement the same projective transformation.

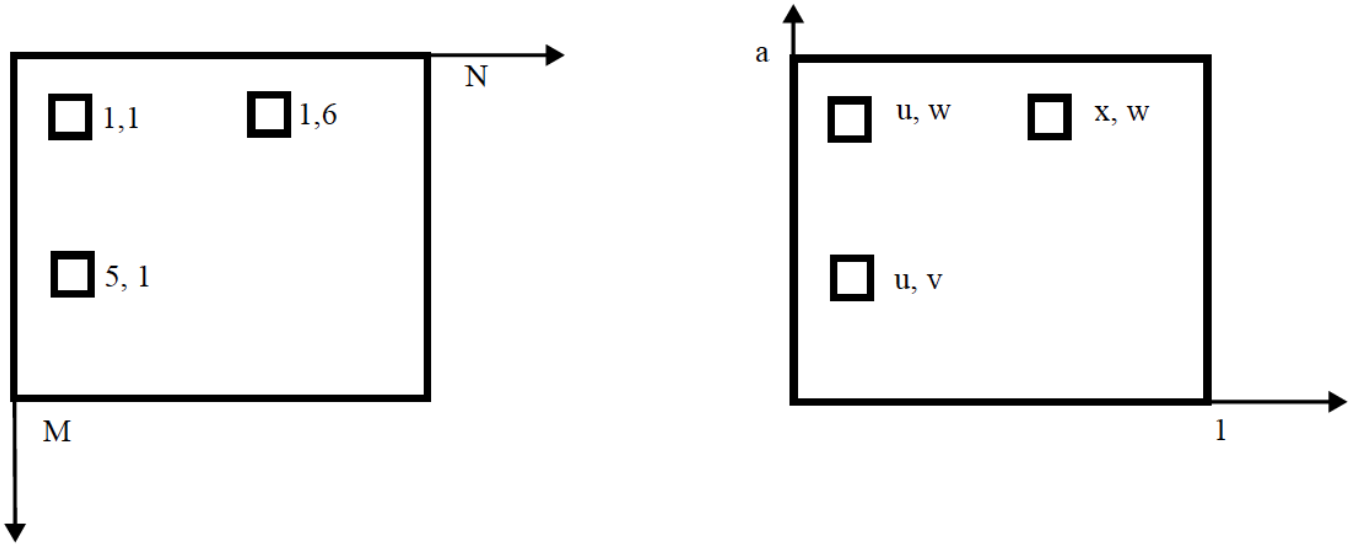
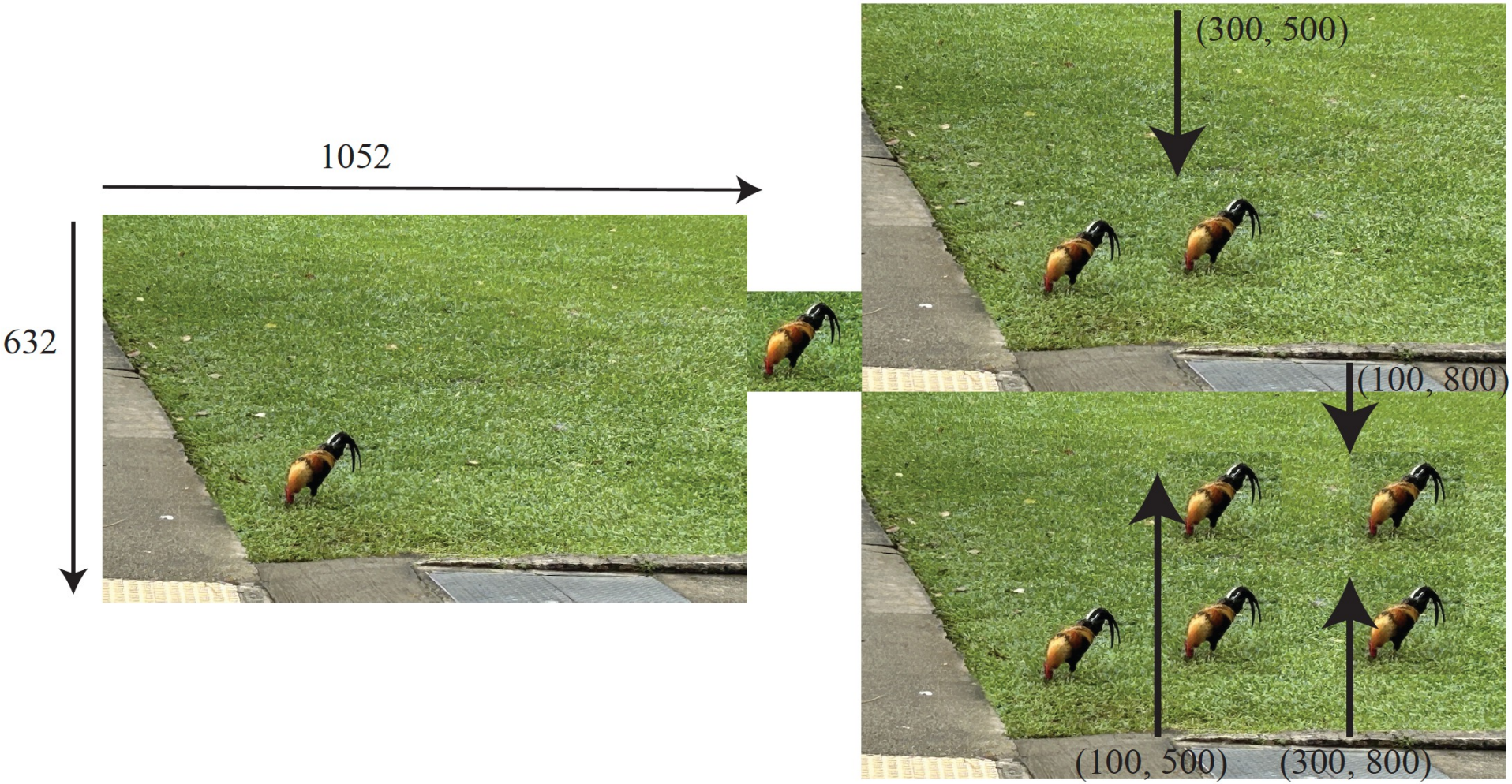


FIGURE 3.3: *The most common coordinate system for images, on the* **left**. *The origin is at the top left corner, and we count in pixels. This is an* $M \times N$ *image. I will use the convention* $\mathcal{I}_{ij}$ *for points in this coordinate system, so the top right pixel is* $\mathcal{I}_{1M}$. *It is usual for pixel locations to be indexed starting at* 1 *(so* $1 \leq i \leq M$ *and* $1 \leq j \leq N$*). In some environments (notoriously, Python), the index starts at* 0. *Keep track of this point, or you will lose some pixels. On the* **right**, *the origin is at the bottom left, and the coordinate axes are more familiar. It is a good idea to use a range from* $0 - 1$ *(rather than* $0 - M$*) in this coordinate system, but if the image is not square one direction will run from* 0 *to* a. *Converting from one coordinate system to the other is straightforward,* but *not being consistent about the coordinate system you are working in is an important source of simple, annoying errors. I will always work in the coordinate system shown on the* **left**.


1052
632
(300, 500)
(100, 800)
(100, 500)
(300, 800)

Blending

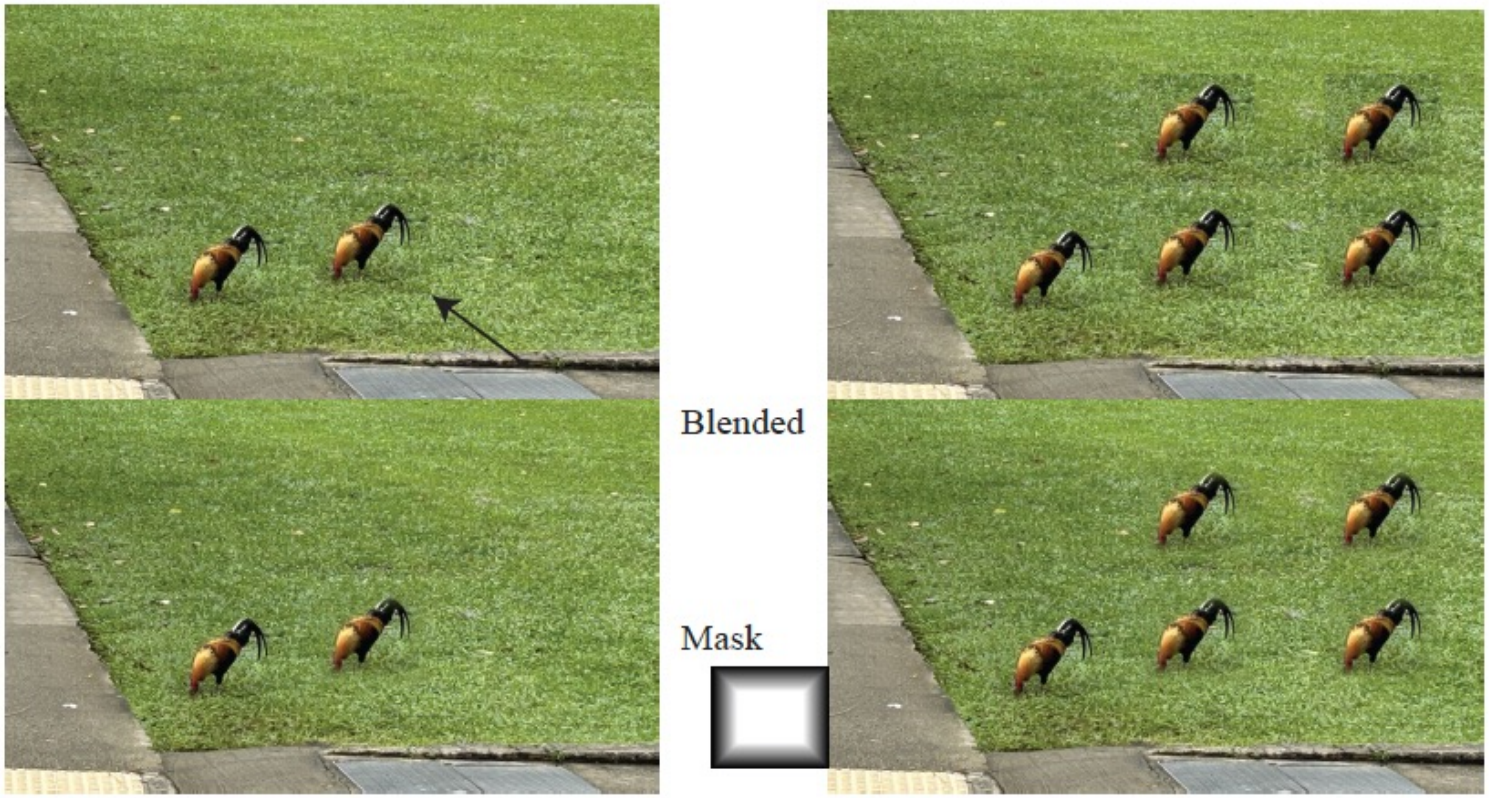
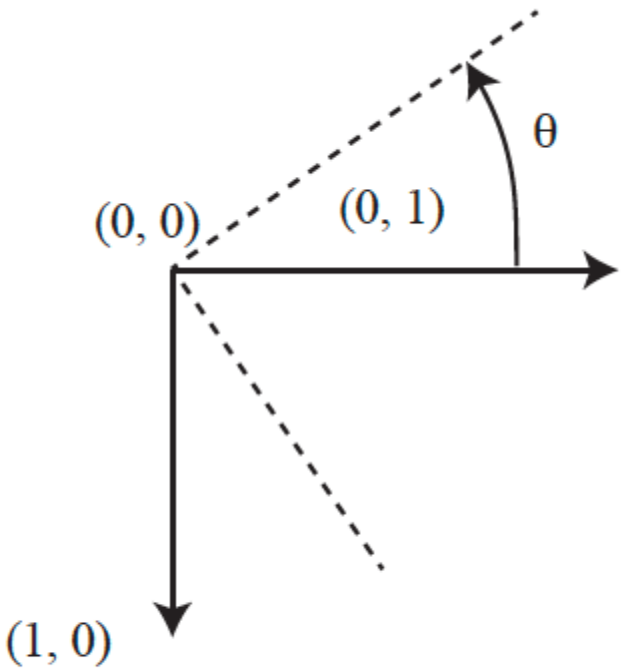
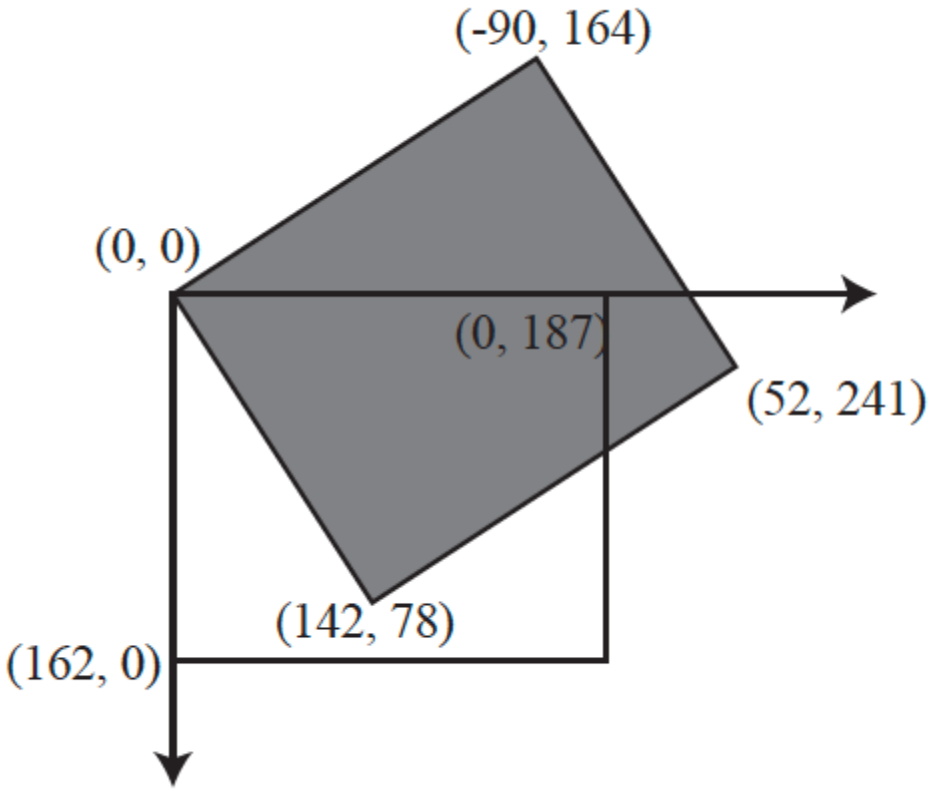


FIGURE 3.5: *The chickens of Figure 3.4 are simply pasted in the* **top row** *(as in that figure, reproduced here for comparison; the arrow on the* **left** *shows a problem with pasting not identified in that figure). In the* **bottom row**, *the chickens have been blended using the blending mask shown. Note the pasting is much less obvious.* Image credit: *Figure shows my photograph of jungle fowl in Singapore.*


θ
(0, 0)
(0, 1)
(1, 0)


(-90, 164)
(0, 0)
(0, 187)
(52, 241)
(142, 78)
(162, 0)

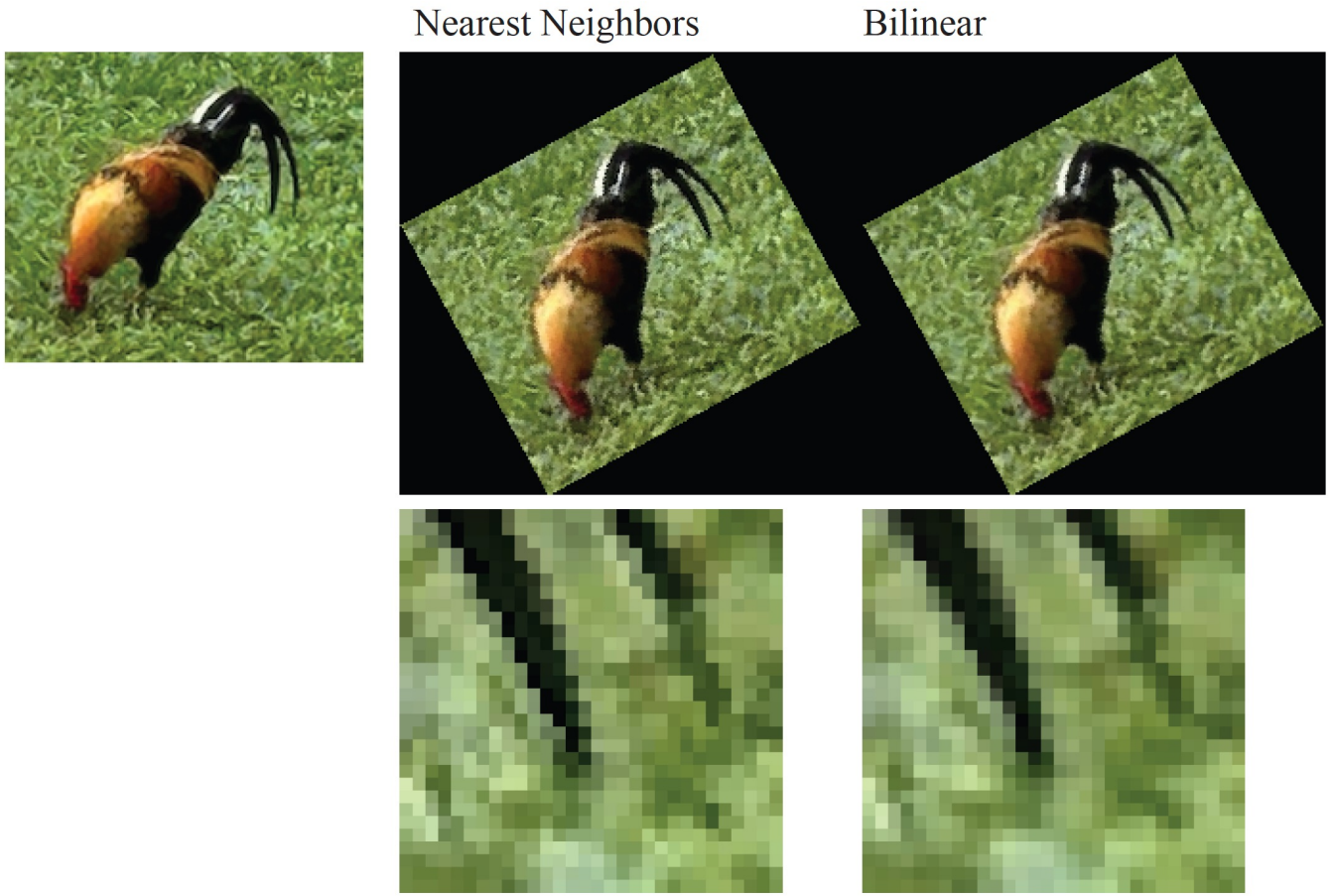
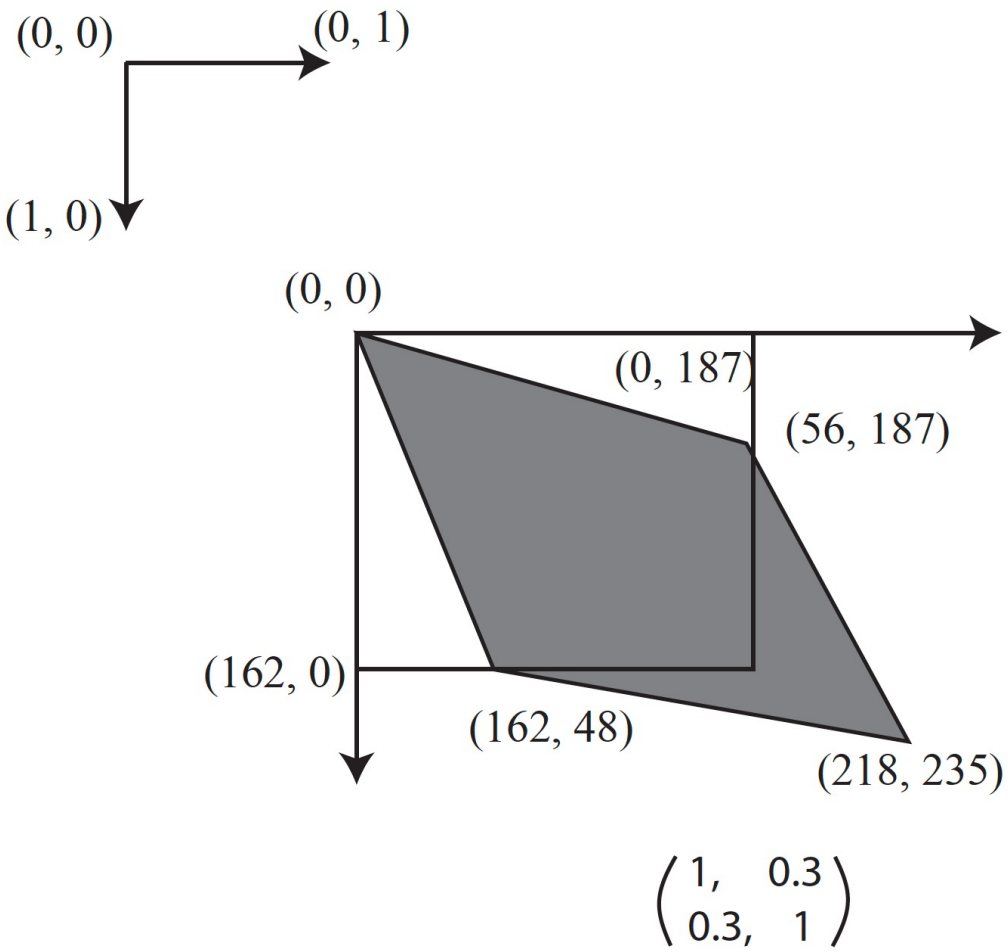
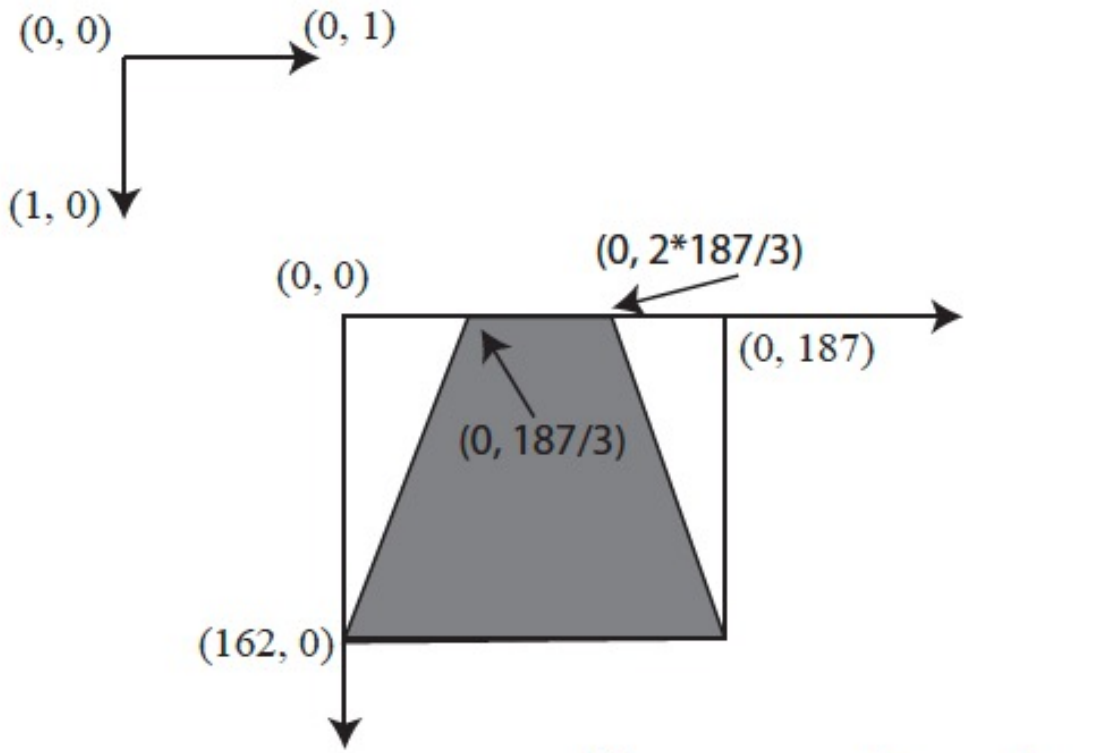


FIGURE 3.7: *The chicken of Figure 3.4, rotated by 0.5 radians as in Figure 3.6, showing the effect of different choices of interpolation. I have zoomed in on a section of the tail feathers to make the difference more apparent.* Image credit: *Figure shows my photograph of jungle fowl in Singapore.*



Nearest neighbors

Bilinear

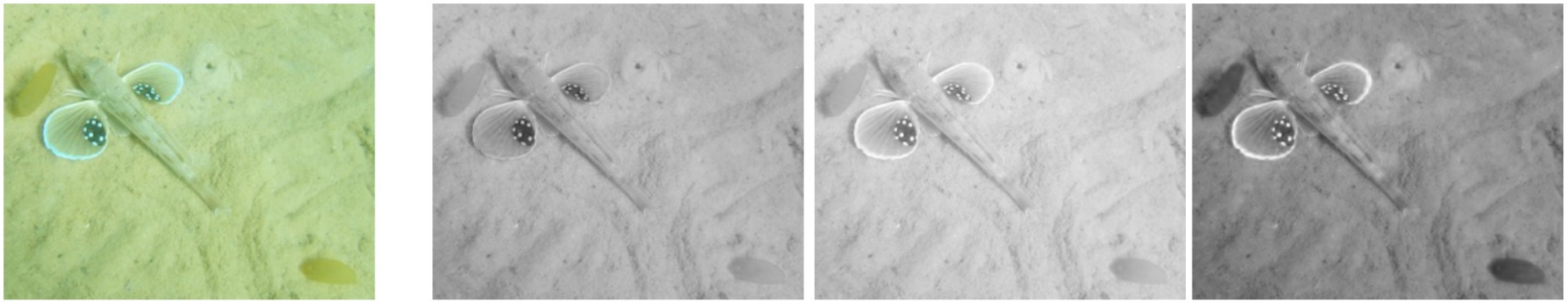


$$\begin{pmatrix} 1/3, & 0, & 0 \\ -187/(3*162), & 1/3, & 187/3 \\ -2/(3*162), & 0, & 1 \end{pmatrix}$$

Nearest neighbors

Bilinear

Image separations



Registering separations

- What happens if R, G, B aren’t exactly aligned?
- (historical photographs)

- Fix R
- Slide G by tx, ty, compute match cost
- Choose tx, ty with best match
- Do B same way

- Q: what match cost?
- Q: what if there are many translations?

Match Costs

- SSD (sum of squared differences)

The *sum of squared differences* or *SSD* scores the similarity between the overlapping parts of two separations $\mathcal{R}$ and $\mathcal{B}$. Given an offset m, n, the SSD is

$$C_{\text{reg}}(m, n; \mathcal{R}, \mathcal{B}) = \frac{1}{N_o} \sum_{\text{overlap}} (\mathcal{R}_{ij} - \mathcal{G}_{i-m,j-n})^2 .$$

Here overlap is the rectangle of pixel locations with meaningful values for both $\mathcal{R}$ and $\mathcal{G}$ and N_o is the number of pixels in that rectangle. Notice that overlap and so N_o change with m and n, so we must compare overlaps of different sizes for different offsets. This means it is important that C_{reg} is an average.

Match Costs

- Cosine distance
  - The *cosine distance*, given by:

$$C_{cos}(m,n) = \sum_{\text{overlap}} \frac{(\mathcal{A}_{ij} * \mathcal{B}_{i-m,j-n})}{\sqrt{\sum_{\text{overlap}} \mathcal{A}_{ij}^2} \sqrt{\sum_{\text{overlap}} \mathcal{B}_{i-m,j-n}^2}}.$$

Annoyingly, this cost function is largest when best, even though it's called a distance. Some authors subtract this distance from one (its largest value) to fix this.

Match Costs

- The *correlation coefficient*, given by:

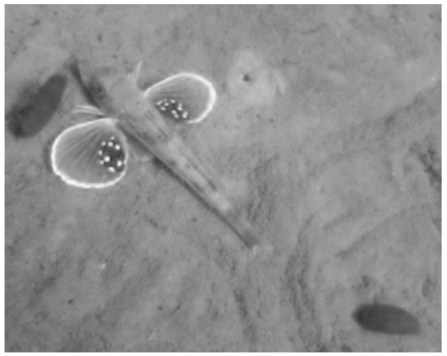
$$C_{corr}(m,n) = \sum_{\text{overlap}} \frac{(\mathcal{A}_{ij} - \mu_A) * (\mathcal{B}_{i-m,j-n} - \mu_B)}{\sqrt{\sum_{\text{overlap}} \mathcal{A}_{ij}^2} \sqrt{\sum_{\text{overlap}} \mathcal{B}_{i-m,j-n}^2}}$$

$$\text{where } \mu_A = \frac{1}{N_O} \sum_{overlap} \mathcal{A}_{ij} \text{ and}$$

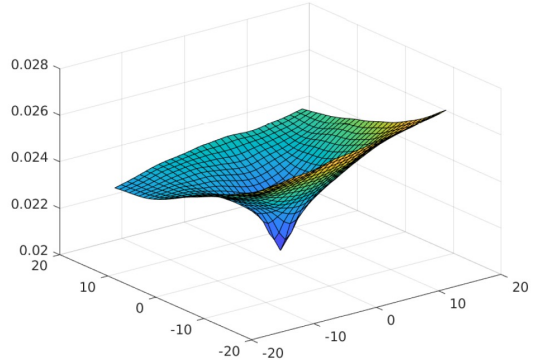
$$\text{where } \mu_B = \frac{1}{N_O} \sum_{overlap} \mathcal{B}_{ij}.$$

This is big for the best alignment. Notice how this corrects for the mean of the overlap in each window.

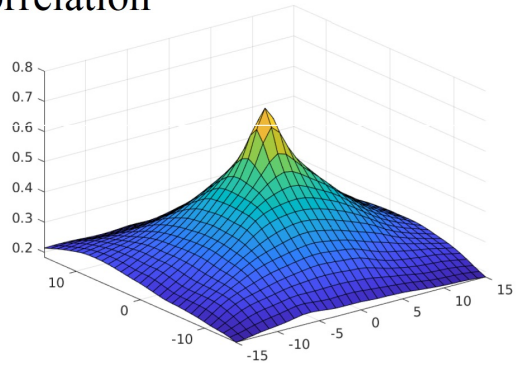
The cost surfaces



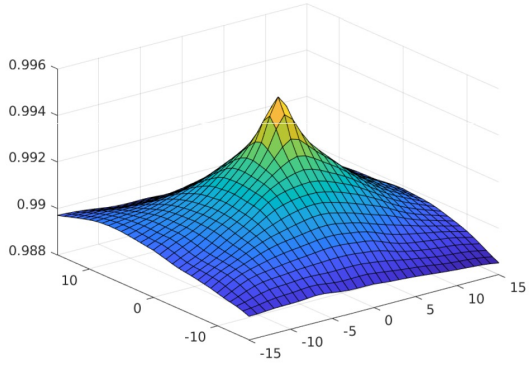
Squared error



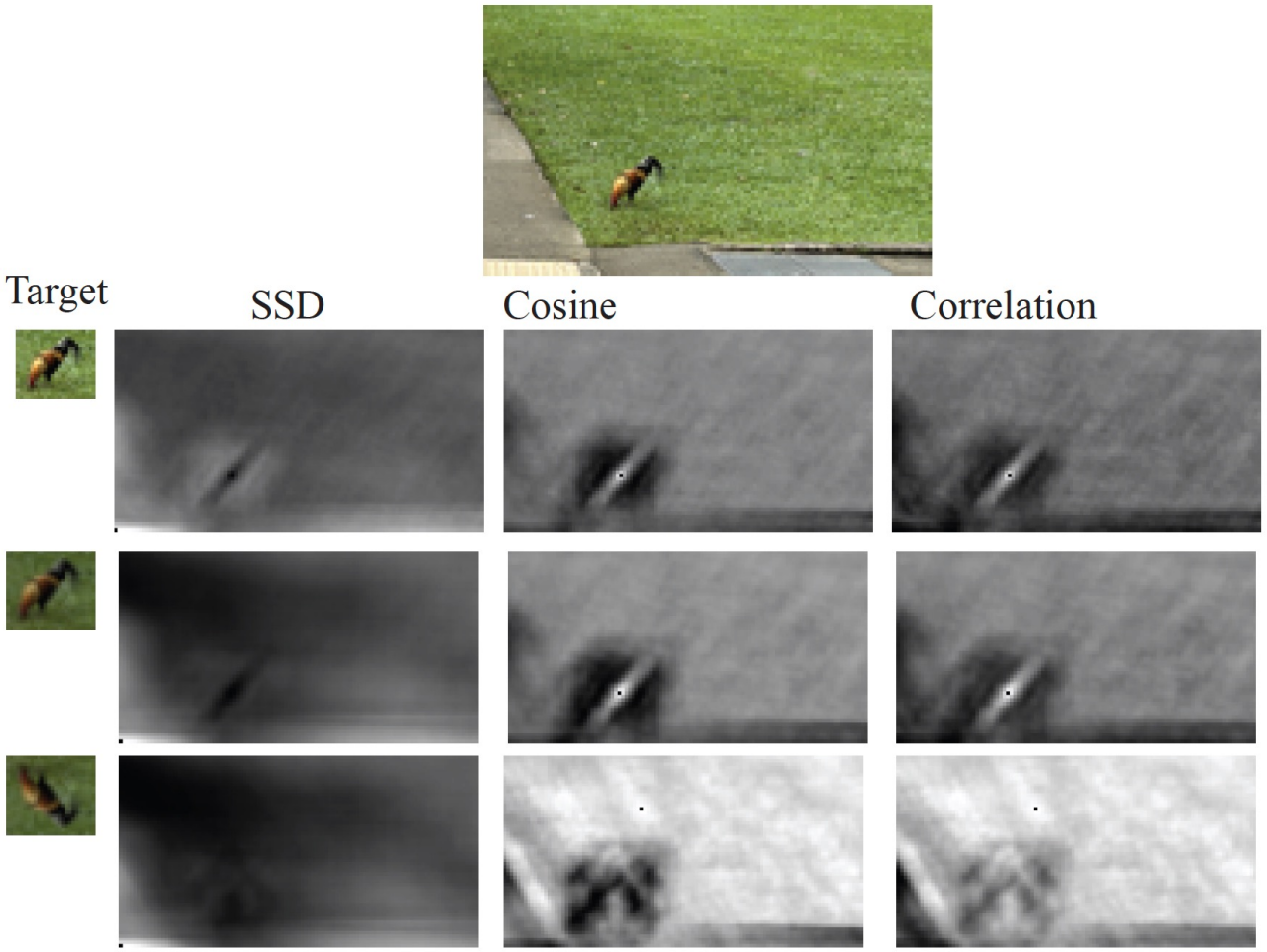
Correlation



Cosine distance



Lagni



What if there are many translations?

- Build a gaussian pyramid (for example, 2x downsampling)
- Align coarsest (N)'th scale by tx, ty
- This gives estimate 2 tx, 2 ty for (N-1) scale translation
- AND search range (-1, 0, 1) x (-1, 0, 1)
- search to get tx', ty'
- Redo for N-2'th scale, etc.

- Very powerful pattern: Coarse to fine search