Digital Image Processing
Fast Pixel Neighborhood Processing in Octave and MATLAB

Tony Richardson
richardson.tony@gmail.com

Several image processing methods require accessing the neighbors of a given pixel. Although this can be done by looping over all the pixels in an image this is a very slow when using Octave or MATLAB. It can be *much* faster to use Octave/MATLAB indexing methods to access all neighbors of a *desired set of pixels* in a single line of code and avoid loops entirely. A method for doing that is described here. If you want to process *all pixels* instead of just a smaller subset of pixels I would recommend that you use the Octave/MATLAB **colfilt** routine. Octave/MATLAB also provide **nlfilter**, but that routine loops over all pixels and is much slower than **colfilt**.

The technique described here will assume a uint8, grayscale image. It is relatively easy to extend to other data types (uint16, double, etc) and formats (binary, RGB color).

We can avoid having to treat the image edges as special cases by working with padded images. Let **a_img** be a 2-D gray-level image of size MxN. We can create a padded image (**a_imgp**) with a one-pixel border by duplicating the edges:

```
a_imgp = [  a_img(1, 1)    a_img(1, :)    a_img(1, end);
            a_img(:, 1)    a_img          a_img(:, end);
            a_img(end, 1)  a_img(end, :)  a_img(end, end) ];
```

Other types of padding, e.g. zero-padding, can be accomplished similarly. A one-pixel border will allow us to access the eight nearest neighbors of all pixels in the original image. If access to larger neighborhoods is desired additional padding will be needed.

Next we create a logical index matrix (**idx**) that can be used to select the desired pixel set in the original (unpadded) image. For example, to select all pixels with values between 16 and 31:

```
idx = (a_img>15 & a_img<32);
```

If you need to randomly select an approximate percentage (**p**) of pixels in the original matrix, try this:

```
idx = rand(M, N)<(p/100);
```

idx will be a logical (binary) matrix with the same dimensions as **a_img**. **idx** will contain ones at the desired pixel locations and zeros elsewhere. **a_img(idx)** will be a column *vector* of the desired pixels. We pad **idx** with logical zeros (to create **idxp**) so that it is the same size as **a_imgp**:

```
idxp = [zeros(1, N+2, ‘logical’);
        zeros(M, 1, ‘logical’)  idx  zeros(M, 1, ‘logical’)
        zeros(1, N+2, ‘logical’)];
```

a_imgp(idxp) should return the same set of pixels as **a_img(idx)**. **idxp** should not select any pixels from the padding.

To access the neighbors of a pixel it is easier to work with the row and column subscripts of the pixel than with the logical index matrix. First form two matrices (RP and CP) that contain the row and column indices respectively of all elements in the padded matrix. The **meshgrid** function can be used to generate these matrices:

```
[CP   RP] = meshgrid(1:NP, 1:MP;
```

Note the order of the arguments and the return values. The first values correspond to the number of columns in the padded matrix, not the number of rows. M and N are the number of rows and columns in the original matrix while MP=M+2 and NP=N+2 are the corresponding values in the padded matrix.

We now use our logical matrix to select the row and column indices (ri_p and ci_p) of all desired pixels in the padded matrix:

```
ri_p = RP(idxp);   ci_p = CP(idxp);
```

You might think that **a_imgp(ri_p, ci_p)** would return the desired set of pixels. It does not. It returns the pixels corresponding to each row index in **ri_p** paired with each column index in **ci_p**. For example, if there are **K** pixels in the desired set, then **ri_p** and **ci_p** will be both be **K** element column vectors. **a_imgp(ri_p, ci_p)** will not return the desired **K** pixels but rather a **K**x**K** matrix (although the desired pixels will be along the diagonal).

You could loop over the ri_p and ci_p vectors to process the selected pixels, but we can avoid the loop by using *linear* indexing. If we use a single index as in **a_imgp(ind)** then **a_imgp** is treated as a column vector (with all the columns stacked on top of each other – Fortran storage) with **ind** selecting an element from the vector. The function **sub2ind** can be used to convert from row and column subscripts to linear indices.

We are now ready to put all of the pieces together. Here are two examples. To replace all of the selected pixels with the pixel to the left of the selected pixel using:

```
a_img(idx) = a_imgp(sub2ind([M+2 N+2], ri_p, ci_p-1));
```

Use ci_p+1, ri_p-1, ri_p+1 instead to select a pixel to the right, below or above a pixel in the selected set. Note that this single line of code replaces ALL selected pixels in the original image (**a_img**) with their left neighbors from the padded image without using a loop. By using a padded image we don't have to worry about edge effects (values in ci_p-1 will always be greater than or equal to one and therefor be valid indices).

To replace all of the selected pixels with the average of its 4-neighbors:

```
a_img(idx) = mean([a_imgp(sub2ind([M+2 N+2], ri_p-1, ci_p))';
                   a_imgp(sub2ind([M+2 N+2], ri_p, ci_p-1))';
                   a_imgp(sub2ind([M+2 N+2], ri_p+1, ci_p))';
                   a_imgp(sub2ind([M+2 N+2], ri_p, ci_p+1))'], 'native');
```

Here each of the individual neighbors of all the desired pixels are transposed into row vectors and stacked on top of each other. Each column of the resulting matrix consists of the four neighbors of one of the selected pixels. The **mean** function will compute the mean of each column. The 'native' argument will cause the function to return a uint8 data type. (It can be omitted here. A double type will be returned, but that will be converted to uint8 by the assignment.)

The method described above works well when only selected pixels are to be processed. If you want to process all pixels you can form an image vector with (for example) 3x3 neighborhoods in columns for all pixels using:

```
img = a_imgp(:);
img_array = [ circshift(img, MP+1)';    circshift(img, MP)';    circshift(img, MP-1)'; …
              circshift(img, +1)';      img';                   circshift(img, -1)';   …
              circshift(img, -MP+1)';   circshift(img, -MP)';   circshift(img, -MP-1)' ];
b_imgp = reshape(median(img_array), MP, NP);
```

I would recommend using **colfilt** in this case though.

Notes:

1. If a neighbor of a selected pixel is also a selected pixel it will be used in the mean calculation. If selected pixels should be excluded you can replace the selected pixels with NaN (Not a Number) and use nanmean instead. There is no uint8 NaN representation, so the padded matrix will need to be converted to type double first. The assignment will convert the values back to uint8.
2. Octave/MATLAB use different rules for handling integer data types than C does. This can be a source of confusion. For example, if x is of type uint8 then z = x + double(255) will be equal to 255 and z will be of type uint8. Similarly if y is of type double then z = x + y will return values between 0 and 255 with z of type uint8. While C promotes data-types in an expression to the highest precision type, Octave/MATLAB demotes to the smallest data type.
3. You can avoid using padded images by skipping processing of any selected pixels that are on any of the image edges. This can be achieved by ANDing the **idx** logical matrix with another logical matrix that has zeros on the edges and ones in the interior.

4. The RP and CP matrices can also be formed using outer products. This can be slightly faster than **meshgrid**.

```
RP = (1:(M+2))’ * ones(1, N+2);
CP = ones(M+2, 1) * (1:(N+2));
```