Write a script (`pr08.m`) that reproduces Figure 5.14. You should use the image

Fig0514(a)(ckt_saltpep_prob_pt25).tif

as your input image and then produce the (b) and (c) figures by processing the input image.

Use the `ordfiltln` function from the image processing package to produce figure (b). Note that the median processing is done on a 7 x 7 neighborhood.

Write a function named `xx_imadapt` that accepts an image (gray-level, uint8) as an input argument and then returns an image (gray-level, uint 8). The function should implement the adaptive, noise reduction algorithm in section “Adaptive median filter” (pps 332-335) of the text book. Your function can use the `padarray` function from the image processing package to pad the input image prior to processing. (Since the algorithm can change the size of the neighborhood during processing I suggest padding to support the maximum possible neighborhood size.)

All scripts should write images out as PNG files as in previous project assignments. Turn in a printed copy of all m-files. Email all m-files (as attachments) to richardson.tony@gmail.com. The email subject must contain “EE415 Project 8” and then your name.

Notes:
- My version of a pixel-by-pixel implementation of the adaptive algorithm (figure (c)) takes 90 seconds on my relatively new, mid-level office desktop and over 120 seconds on a 6-year old, mid-level laptop. So that I can be sure that your function is actually running and not stuck in a loop, your function should display its progress as a percentage in 10% increment. You can use “fflush(stdout)” after any display statements to force the display output buffer to be immediately sent to the command window.
- A matrix version of the adaptive algorithm runs in just over a second. Loop over the neighborhood size while using `ordfiltln` to compute min, max, and median values for each pixel. Use logical index matrices or the `find` function to determine which pixels are ready for Stage B processing each time through the loop. (You will need to exclude pixels that have already reached Stage B in a previous loop iteration.) The following code (which uses logical index matrices) may help point you in the right direction:

```matlab
z_min = ordfiltln(a, 1, true(winsize), 'symmetric');
z_max = ordfiltln(a, winsize*winsize, true(winsize), 'symmetric');
z_med = ordfiltln(a, (winsize*winsize+1)/2, true(winsize), 'symmetric');
A1 = z_med - z_min;
A2 = z_max - z_med;  % This should always be positive or zero
% Find all of the new Stage Bs (Exclude pixels that have already reached Stage B)
new_stage_b(:) = 0;
new_stage_b(~stage_b) = and(A1(~stage_b) > 0, A2(~stage_b) > 0);
% Keep track of all pixels that have reached Stage B.
stage_b = or(stage_b, new_stage_b);
```

This code would be inside the neighborhood resizing loop. In this code, `a` is the input matrix, `winsize` is the neighborhood size (`winsize` is initially 3 and increases by 2 with every loop iteration), `stage_b` and `new_stage_b` are logical matrices that are the same size as `a` and are created prior to entering the loop. The new Stage B pixels are then split into two groups (according to the algorithm) for further processing. Additional logical index matrices may be useful for this split.