```
%% Note: blkproc was deprecated when blockproc was introduced in R2009b
% The new blockproc function supports file-based block processing for
% arbitrarily large TIFF images. The new function supports in-memory
% operations as well as file-to-file processing of images which are too
% large to load completely into memory.

%% Note: Submission has been restructured using encoder/decoder stages
% My goal was to demonstrate the similarities between 9.8 and 9.9, and to
% also show how this assignment fits into the larger process of image
% encoding and decoding. The actual bitstream coding stages, however, are
% naturally left empty for now.

%% Main program
function newton_joshua_a03
    %% Load image
    [file,path] = uigetfile('*.tif'); % Returns filename and path to file
    input_img = imread(strcat(path,file)); % uint8 grayscale image matrix

    %% Process images using Problem 9.8 specifications
    % Partial set of DCT coefficients, K = [64, 32, 16, 8, 4, 2]
    p98_images = cell(6,1);
    p98_images{1} = a03_JPEG('Problem 9.8', input_img, 64);
    p98_images{2} = a03_JPEG('Problem 9.8', input_img, 32);
    p98_images{3} = a03_JPEG('Problem 9.8', input_img, 16);
    p98_images{4} = a03_JPEG('Problem 9.8', input_img, 8);
    p98_images{5} = a03_JPEG('Problem 9.8', input_img, 4);
    p98_images{6} = a03_JPEG('Problem 9.8', input_img, 2);

    %% Process images using Problem 9.9 specifications
    % Quantized DCT coefficients, scale factors = [0.5, 1, 2, 4, 8, 16]
    p99_images = cell(6,1);
    p99_images{1} = a03_JPEG('Problem 9.9', input_img, 0.5);
    p99_images{2} = a03_JPEG('Problem 9.9', input_img, 1);
    p99_images{3} = a03_JPEG('Problem 9.9', input_img, 2);
    p99_images{4} = a03_JPEG('Problem 9.9', input_img, 4);
    p99_images{5} = a03_JPEG('Problem 9.9', input_img, 8);
    p99_images{6} = a03_JPEG('Problem 9.9', input_img, 16);

    %% Display images for comparison
    figure(1);
    montage(p98_images);

    figure(2);
    montage(p99_images);
end
```

```
function reconstructed_img = a03_JPEG(problem_number, input_img, param)
    %% Stage 1: Block transform
    fun = @(block_struct) round(dct2(block_struct.data));
    dct_coeff = blockproc(input_img,[8,8],fun);

    %% Stage 2: Compression (discarding or quantization of coefficients)
    switch problem_number
        case 'Problem 9.8' % param = number of retained coefficients
            % Create matrix in zigzag order of 1s (retain) and 0s (discard)
            M = zigzag(ones(1,param),8);

            % Create partial set of DCT coefficients
            fun = @(block_struct) double(block_struct.data).*M;
            partial_coeff = blockproc(dct_coeff,[8,8],fun);

        case 'Problem 9.9' % param = scaling factor
            % Generate JPEG normalization matrix
            m = [16 11 10 16 24 40 51 61
                 12 12 14 19 26 58 60 55
                 14 13 16 24 40 57 69 56
                 14 17 22 29 51 87 80 62
                 18 22 37 56 68 109 103 77
                 24 35 55 64 81 104 113 92
                 49 64 78 87 103 121 120 101
                 72 92 95 98 112 100 103 99];

            % Create quantization matrix using scale factor
            QM = m * param;

            % Create partial set of DCT coefficients
            fun = @(block_struct) ...
                  floor( (double(block_struct.data)+(QM/2))./(QM) );
            quantized_coeff = blockproc(dct_coeff,[8,8],fun);
    end

    %% Stage 3/4: Coding/Decoding of coefficients

    %% Stage 5: Dequantization of coefficients
    switch problem_number
        case 'Problem 9.8'
            % No dequantization needed for this method.
        case 'Problem 9.9'
            fun = @(block_struct) double(block_struct.data).*QM;
            partial_coeff = blockproc(quantized_coeff,[8,8],fun);
    end

    %% Stage 6: Inverse block transform
    fun = @(block_struct) idct2(block_struct.data);
    reconstructed_img = uint8(blockproc(partial_coeff,[8,8],fun));
end
```

```
% vector a --> square matrix b (sizeb-by-sizeb) in zigzag order
function b = zigzag(a,sizeb)
    %% Initializing variables
    b = zeros(sizeb,sizeb);
    n = 0;

    %% Ensuring that the # of elements in a and b are equal
    if length(a) < sizeb^2
        a = [a, zeros(1,sizeb^2-length(a))];
    else
        a = a(1:sizeb^2);
    end

    %% Zigzag operations
    for k = 1:sizeb
        n = n+k-1;
        for i = 1:k
            if rem(k,2)==0
                b(i,k+1-i) = a(n+i);
            else
                b(k+1-i,i) = a(n+i);
            end
        end
    end

    for k = 2:sizeb
        n = n+sizeb+1-k;
        for i = k:sizeb
            if rem((sizeb-k),2)==0
                b(k+sizeb-i,i) = a(n+i);
            else
                b(i,k+sizeb-i) = a(n+i);
            end
        end
    end
end
```