CSE803 HW2

Javen W. Zamojcin
zamojci1@msu.edu

2024, October 3

1 Question 1.) Image Filtering

1.1 Part A.) Image Patches

1.1.1

To implement the `image_patches()` function, I simply created a nested-forloop to divide the given image into as many sub-windows of the given dimensions as possible. I also normalize the individual patches in this function by subtracting the mean and dividing by the standard deviation.

(a) Patch0 (b) Patch1 (c) Patch2

Figure 1: Q1-P1 Patches

1.1.2

These patches would not make good image descriptors. The raw image patches are not invariant enough to changes in illumination or viewing geometry. This descriptor variance is what inspired work in invariant feature extraction.

1.2 Part B.) Gaussian Filter

1.2.1

1. Let $G(x)$ and $G(y)$ be horizontal and vertical 1D Gaussian kernels, respectively, and $I(x, y)$ the image matrix.

2. The sequential convolution of the two kernels in one ordering is defined as $(I * G(x)) * G(y)$.

3. By the associative property of convolutions: $(I * G(x)) * G(y) = I * (G(x) * G(y))$.

4. The convolution of two filters is their outer product: $G(x) * G(y) = G(x).G(y)$.

5. $G(x).G(y) = (\frac{1}{\sqrt{2\pi}\sigma} \exp \frac{-x^2}{2\sigma^2}) * (\frac{1}{\sqrt{2\pi}\sigma} \exp \frac{-y^2}{2\sigma^2}) = (\frac{1}{2\pi\sigma^2} \exp \frac{-x^2+y^2}{2\sigma^2}) = G(x, y)$

The convolution of two 1D Gaussian functions with variances σ_1^2 and σ_2^2 produces a 2D Gaussian function with variance $\sigma_1^2 + \sigma_2^2$.

1.2.2

To implement the specified Gaussian kernel, I created a function `gaussian_kernel()` which calculates the kernel for the given length and standard deviation. This uses a standard implementation of creating a Length x Length linearly enumerated matrix, applying the Gaussian equation to each cell and then averaging.

For `convolve()`, I implemented the 'valid' mode approach where the convolution product is only given for points where the signals overlap completely. In other words, where full windows that match the kernel dimensions may be extracted.

The Gaussian filter adds a blurring effect to the image.

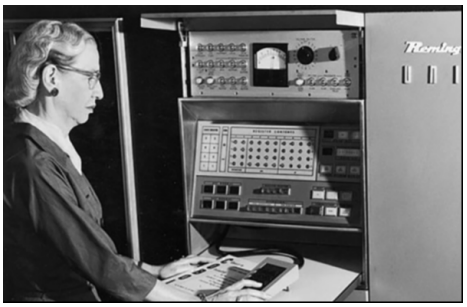


Figure 2: Gaussian Filtered

1.2.3

$k_x = [1, 0, -1]$ / 2
$k_y = [1, 0, -1]^T$ / 2

To complete the function `edge_detection()`, I first calculate the partial derivatives, I_x and I_y, by convolving with their respective derived kernels, k_x and k_y. Then I calculate the gradient magnitude as the square root of the sum of the squared image partial derivatives.

1.2.4

The edge-detect image with the original image as input has much more defined edges or white lines, but also has more noise throughout the entire image. The edge-detect image with the gaussian-filtered image as input has much of this noise removed, but at the cost of having the edges slightly blurred.

Figure 3: Edge Detection

Figure 4: Gaussian Edge Detection

1.3 Part C.) Sobel Operator

1.3.1

1. It holds that $\frac{d}{dx,y}(f * g) = f * \frac{d}{dx,y}g$.

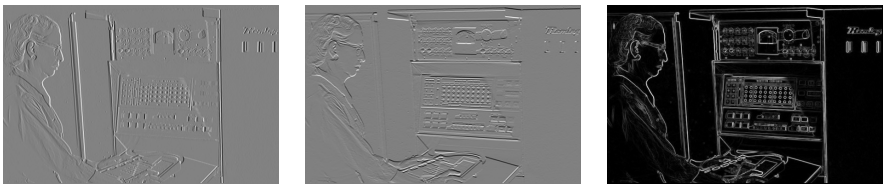
2. $G_{x,y} = I * \frac{d}{dx,y}(K_{Gaussian})$

3. $\frac{d}{dx}(K_{Gaussian}) = \begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix} * \begin{bmatrix}-1 & 0 & 1\end{bmatrix} = \begin{bmatrix}1 & 0 & -1\\ 2 & 0 & -2\\ 1 & 0 & -1\end{bmatrix}$

4. $\frac{d}{dy}(K_{Gaussian}) = \begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix} * \begin{bmatrix}-1\\ 0\\ 1\end{bmatrix} = \begin{bmatrix}1 & 2 & 1\\ 0 & 0 & 0\\ -1 & -2 & -1\end{bmatrix}$

1.3.2

To complete the `sobel_operator()` function, I simply take the convolution of our predefined Sobel kernels, S_x and S_y, and the given image to produce their respective Gaussian-filtered image derivatives, G_x and G_y. The gradient magnitude is the square root of the sum of the squared image derivatives.



(a) Gx (b) Gy (c) Gradient Magnitude

Figure 5: Q1-P3 Sobel Operators

1.3.3

1. $S(I, \alpha) = G_x cos(\alpha) + G_y sin(\alpha)$

2. $S(I, \alpha) = cos(\alpha)(I * K_x) + sin(\alpha)(I * K_y)$

3. $S(I, \alpha) = I(cos(\alpha)K_x + sin(\alpha)K_y)$

4. $S(I, \alpha) = I * K(\alpha)$

To complete the function `steerable_filter()`, for each given angle I first compute the $K(\alpha)$ as $cos(\alpha) * S_x + sin(\alpha) * S_y$, where S_x and S_y are the respective Gaussian kernel derivatives. I then calculate the filtered image for the given angle by convolving the original image with the respective $K(\alpha)$.

These kernels detect edges steered in the gradient orientation given by the angle. In other words, changing the angle will emphasize edges differently according to the gradient direction.

1.4 Part D.) LoG Filter

1.4.1

The first LoG kernel is an instance of a Positive laplacian filter. This filter identifies the outward edges of an image. The second kernel is also a positive laplacian filter but is combined with a Gaussian operation for smoothing or filtering out noise. The edges in this second filtered image appear to be more emphasized, although being more blurry.

(a) LoG Filter 1 (b) LoG Filter 2

Figure 6: Q1-P4 LoG Filtered Images

1.4.2

Consider two Gaussian functions with different variances. The wider one (larger variance) captures the low-frequency components of an image, while the narrower one captures the high-frequency details. The result of their difference highlights regions with a significant change in intensity (edge detection). This produces similar edge-detection capabilities of the second derivative effect of the LoG.

2 Question 2.) Feature Extraction

2.1 Part A.) Corner Score

2.1.1

To complete the function `corner_score()`, I start by creating a zero-padded output matrix with larger dimensions than the given image to account for the non-ideal window comparisons. Then I iterate through each given pixel in a nested for-loop, calculating the first window centered at the pixel origin and the second centered at the given pixel offset. The output is then computed as the sum of squared distances between these windows for each pixel.

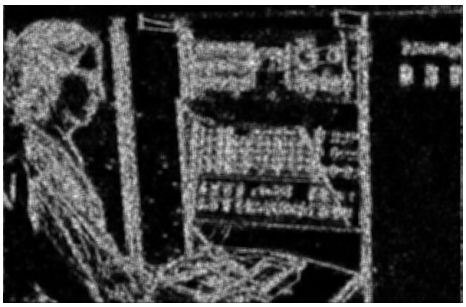


Figure 7: Q2-P1 Corner Score

2.1.2

Using this method and performing an iterative parameter search for the ideal offsets quickly becomes computationally impractical as you scale the image resolution or want to add an additional image dimension. The number and length of nested for-loops becomes very inefficient.

2.2 Part B.) Harris Corner Detector

To complete the function `harris_detector()`, in my final implementation I begin by defining the Sobel kernels again and calculating the Gaussian kernel with a length given by the window size parameter. Then, I compute the filtered-image derivatives, first by convolving the image with the Sobel kernels, and then convolving the squares and product of these derivatives with the Gaussian kernel. Finally, I compute the the determinant and trace matrices from the Gaussian filtered images, construct the structure tensor, calculate the Harris response with an alpha value of 0.05, then apply a binary threshold pass of 0.5.

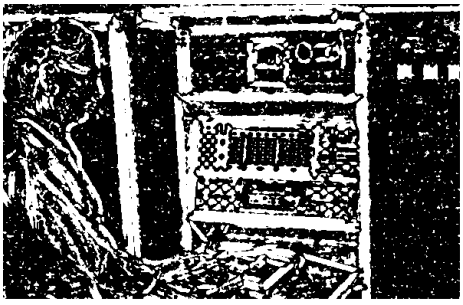


Figure 8: Q2-P2 Harris Corner Response

3 Question 3.) Blob Detection

3.1 Part A.) Single-scale Blob Detection

3.1.1

To complete the function `gaussian_filter()`, I simply computed a Gaussian kernel with a length of the given kernel-size variable and sigma parameter. Then I use the *cv*2 library to perform the kernel convolution just to ensure the implementation accuracy. I also implemented a `difference_of_gaussian()` function for this step which accepts an image and two sigma parameters, calculates their respective Gaussian filters, and takes their difference.

3.1.2

For the small dot filter, I chose sigma values $8.0, 11.3$. This resulted in about 25 maxima (yellow peaks). However, the small dot reactivity was not as clearly defined as the large dots, and blurred partially with the filler space. For the large dot filter, I chose sigma values $22.6, 32.0$. This resulted in 16 clearly defined maxima, matching well with the large dots. See Figure 9.

3.2 Part B.) Scale Space Representation

3.2.1

To complete the function `scale_space()`, I first initialize a 3D scale space image matrix, with S-1 for the 3rd dimension. Then I iterate for S-1 times, calculating σ_1 as the product of the minimum sigma and the k parameter to the ith power, and σ_2 to the power of $i+1$, then taking their difference of Gaussian and storing the resulting image in the output array.

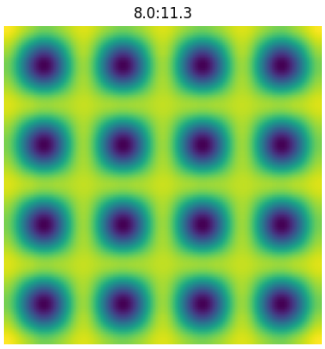
3.2.2

For both scale space generations, I used parameters $S = 8$, $k = \sqrt{2}$, and the respective smallest sigma values as the initial values. The maxima are not clearly defined at every scale, for both the large and small dots. Some scales, such as $16.0 : 22.6$ blur both dot types. See figure 10.

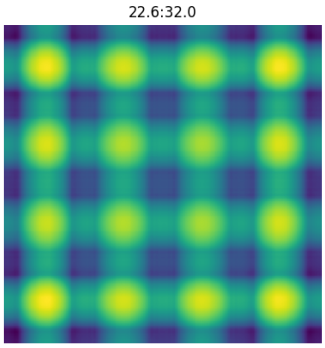
3.3 Part C.) Blob Detection

Figure 11 displays the visualized maxima using the sigma values chosen for part 1: $8.0, 11.3$ for the small dots and $22.6, 32.0$ for the large dots. These values resulted in a good prediction of their respective target dots, matching 25/25 small dots and 16/16 large dots, with no false peaks as far as I can tell. For the k_xy and k_s parameters, I simply used the default values of $k_xy = \sigma_{small}$ and $k_s = 1.0$ for the small filter, and $k_xy = 10$ and $k_s = 1.0$ for the large filter.

These parameters affect false positive detection by checking the surrounding



(a) Polka Small DoG



(b) Polka Large DoG

Figure 9: Q3-P1 Gaussian Filter Responses

pixels to verify a peak isn't just a local peak. Setting this to be too large will result in few or none peaks being found, and the opposite effect of too many peaks if set too low.

3.4 Part D.) Cell Counting

Figure 12 displays the four raw cell images I randomly selected as well as their preprocessed versions. For preprocessing, I applied a simple binarization threshold of 0.07 to remove noise, which significantly helped cell detection.

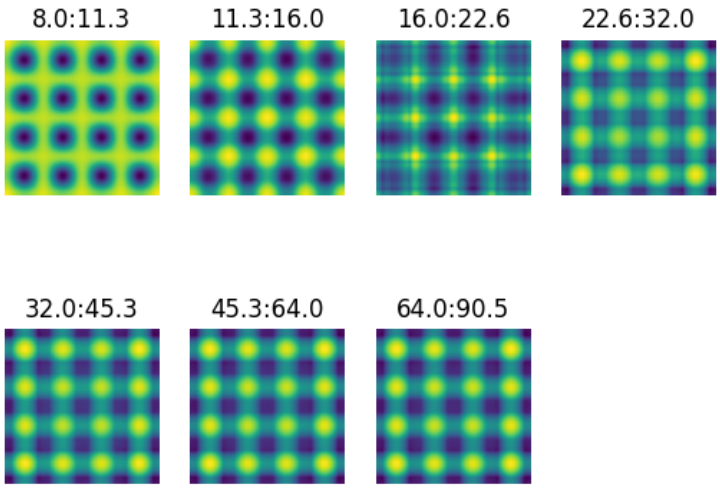
Figures 13-16 display the detected cells visualized across a scale space, the parameters used and the resulting cell count, for each cell image. All four cell images used maxima parameters $k_x y = 10$ and $k_s = 1.0$, and scale space parameters $S = 8$, $k = \sqrt{2}$, and $\sigma_{start} = 1.1$.

For 031-cell.png, the best parameters both in terms of cell count and centering quality were $\sigma_1 = 2.2$, $\sigma_2 = 3.1$, resulting in $cellcount = 116$.

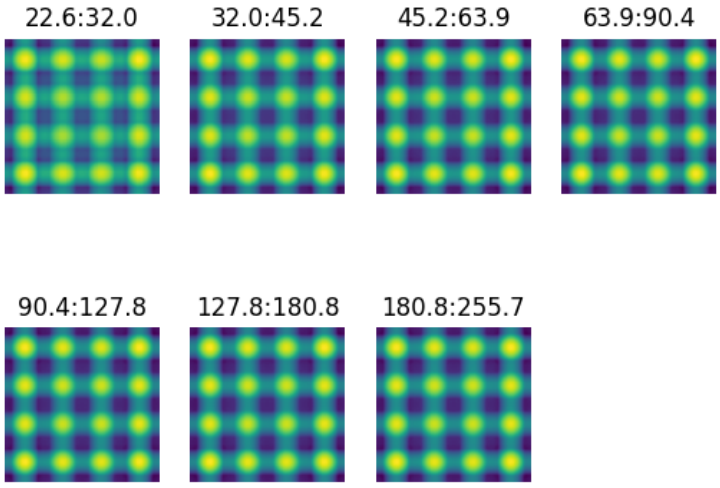
For 054-cell.png, the best parameters in terms of just cell count were $\sigma_1 = 1.1$, $\sigma_2 = 1.6$, resulting in $cellcount = 126$. However, these maxima were not very centered despite having better individual cell detection.

For 073-cell.png, the best parameters were again $\sigma_1 = 2.2$, $\sigma_2 = 3.1$, resulting in $cellcount = 106$. This was both the highest cell count and had fairly centered maxima.

For 106-cell.png, the best parameters for cell count were $\sigma_1 = 1.6$, $\sigma_2 = 2.2$, resulting in $cellcount = 80$. This tied with $\sigma_1 = 1.1$, $\sigma_2 = 1.6$, but had slightly better maxima centering.

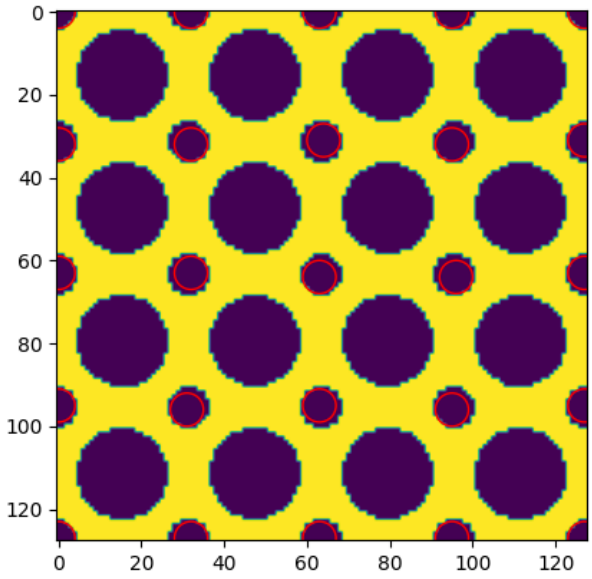


(a) Scale Space Small

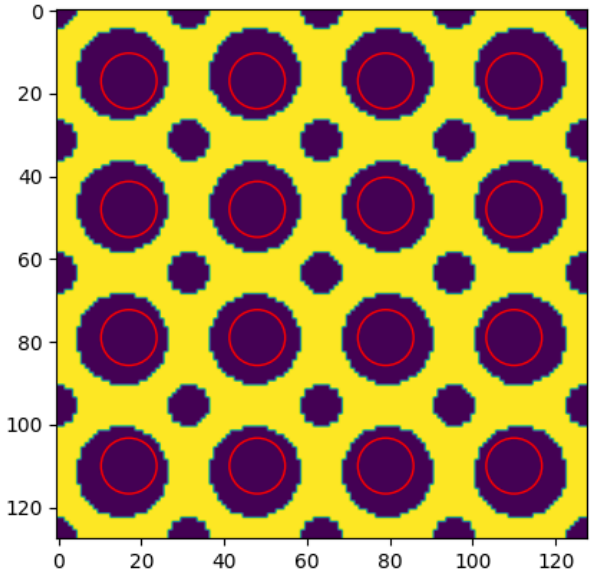


(b) Scale Space Large

Figure 10: Q3-P2 Scale Space Responses

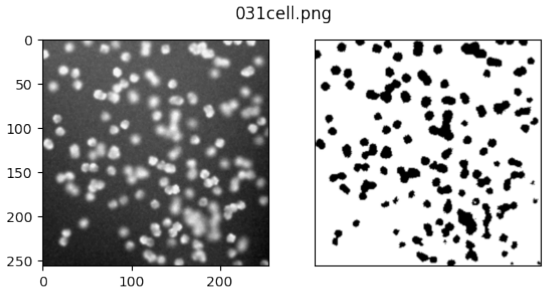


(a) Polka Small Maxima

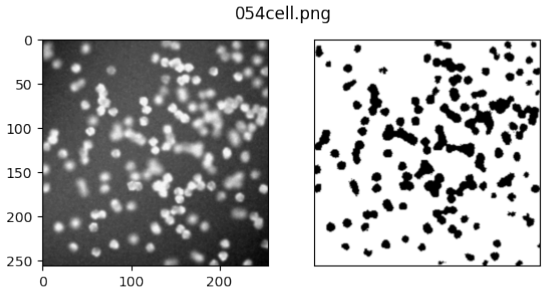


(b) Polka Large Maxima

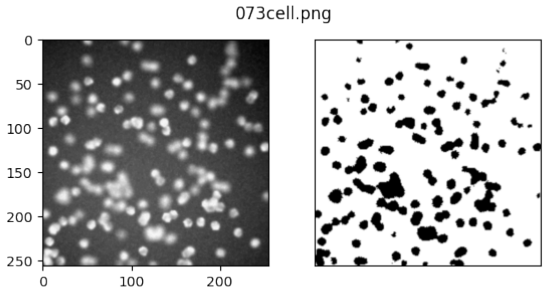
Figure 11: Q3-P3 Gaussian Filter Maxima



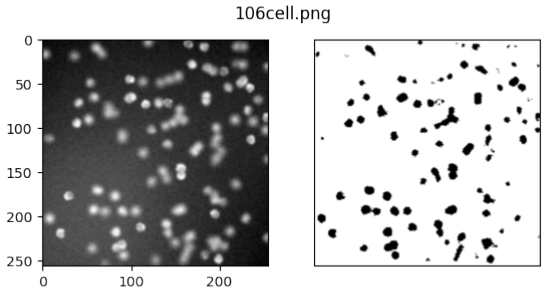
(a) Preprocessed 031-cell



(b) Preprocessed 054-cell



(c) Preprocessed 073-cell



(d) Preprocessed 106-cell

Figure 12: Q3-P4 Preprocessed Cell Images

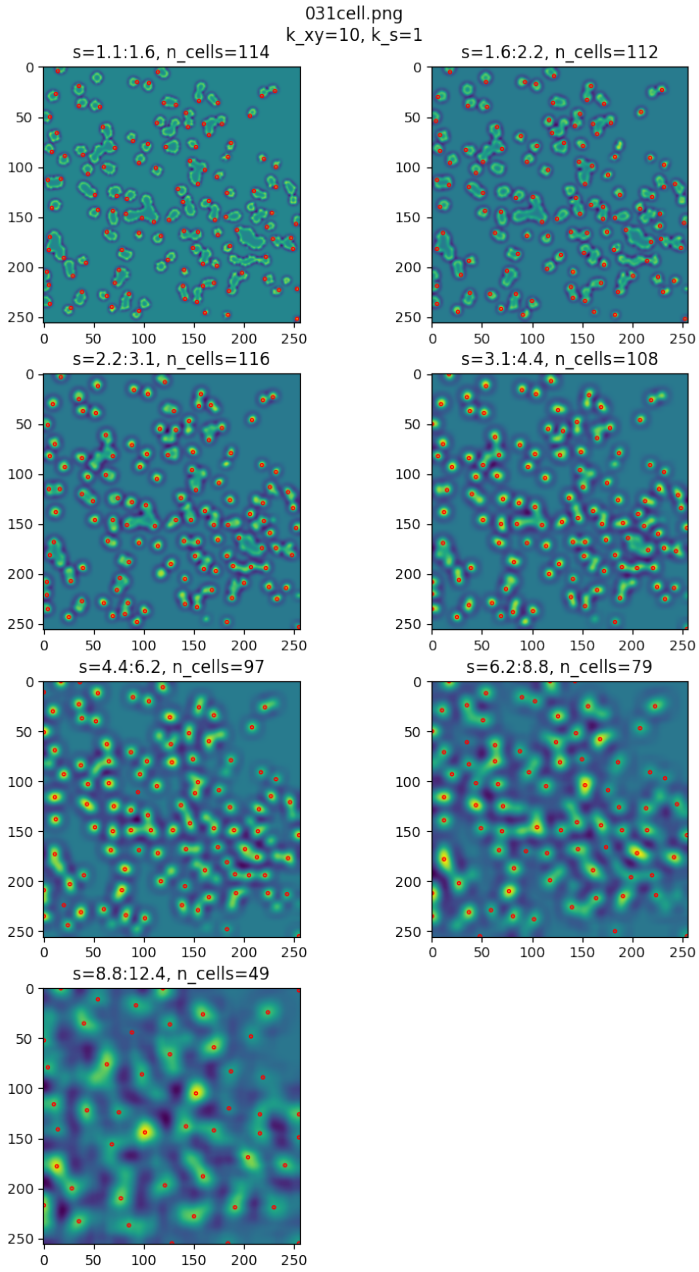


Figure 13: Scale Space Maxima 031-cell

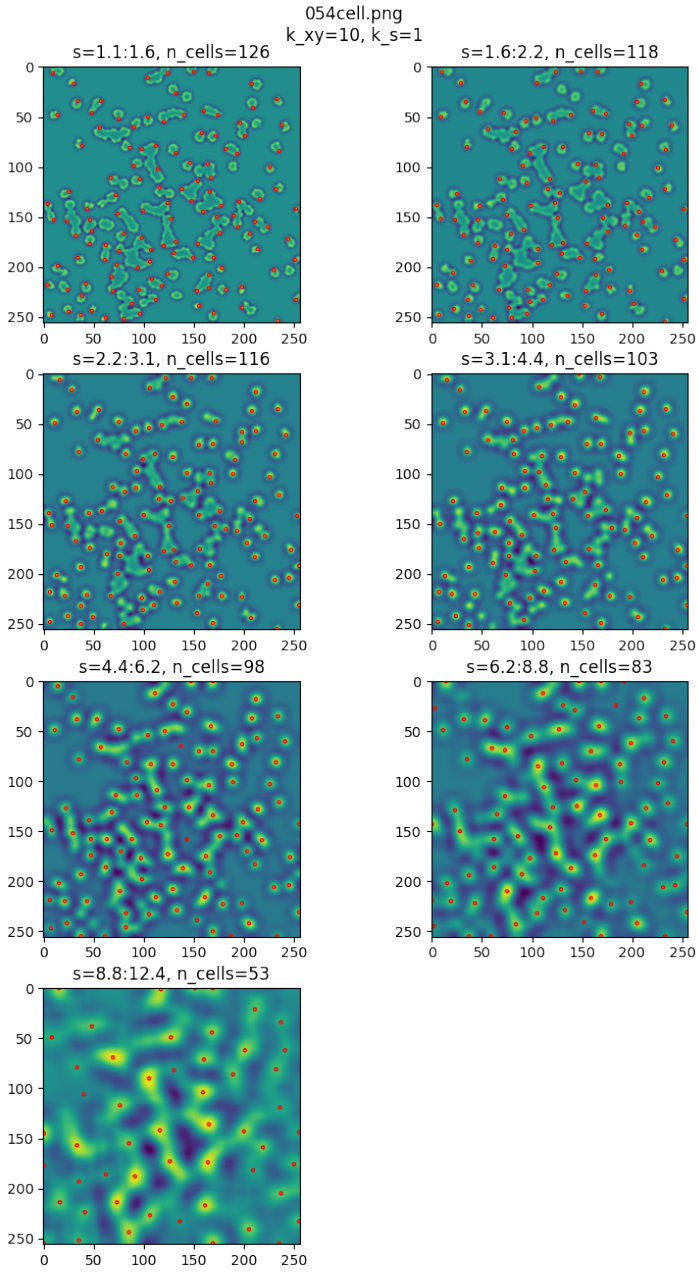


Figure 14: Scale Space Maxima 054-cell

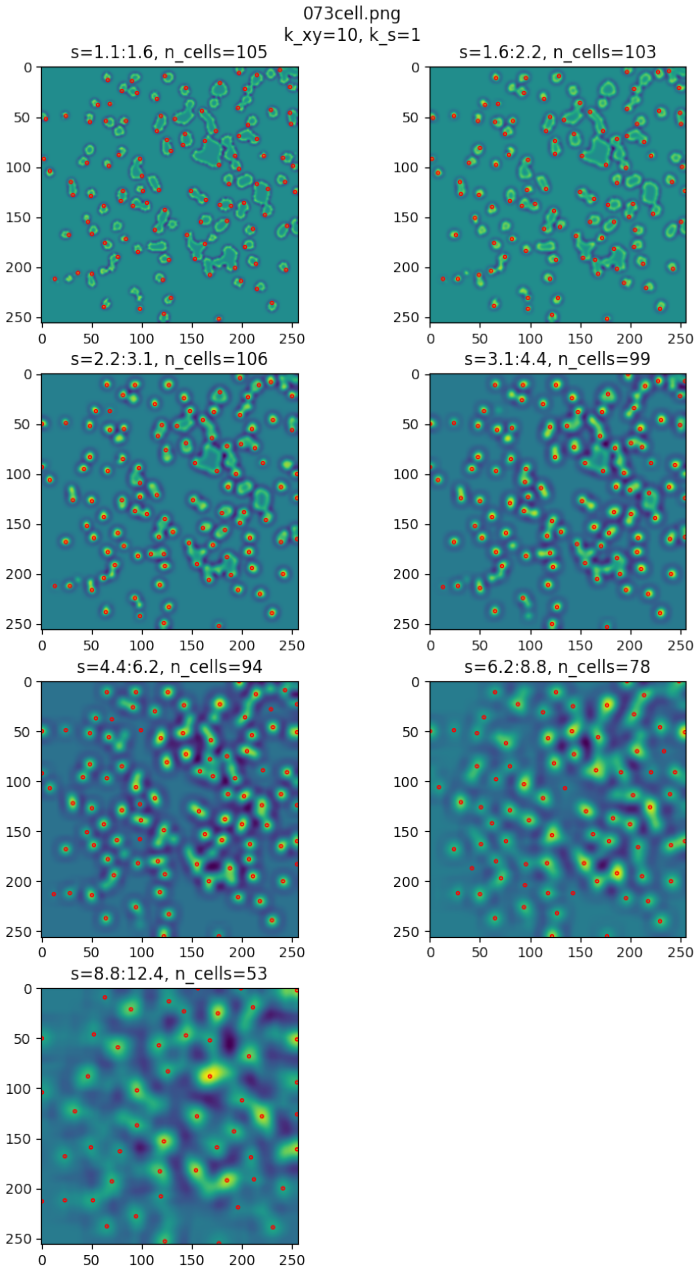


Figure 15: Scale Space Maxima 073-cell

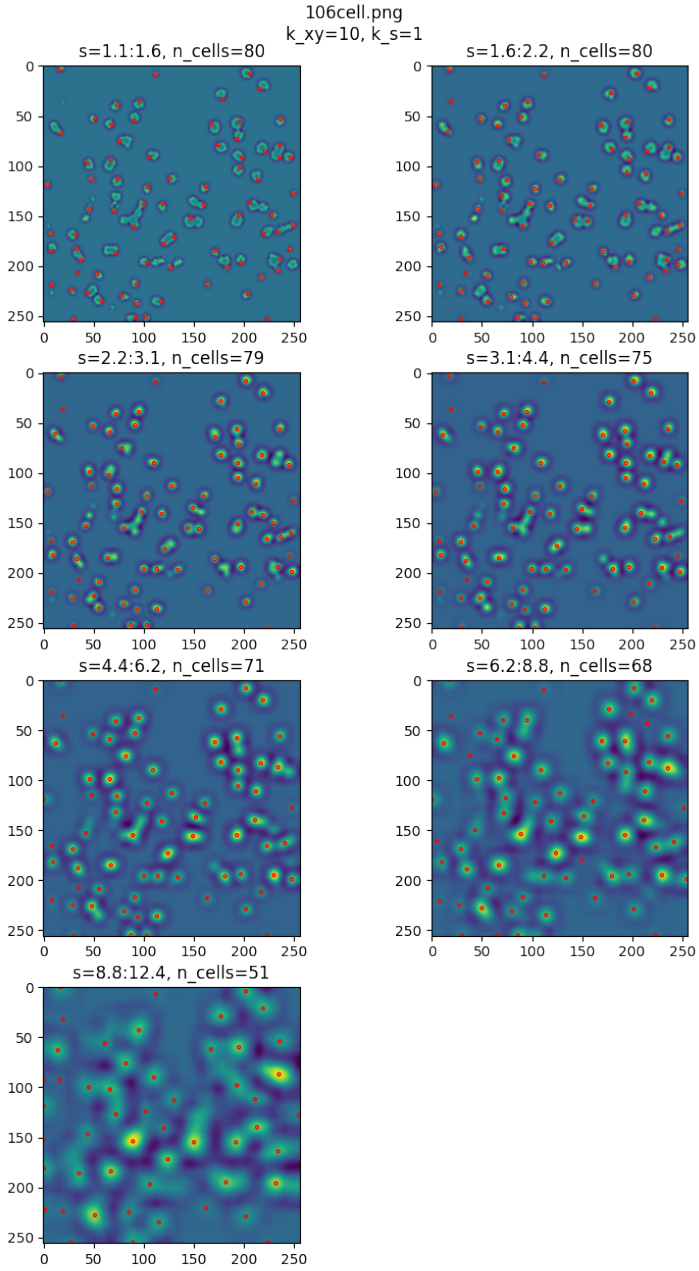


Figure 16: Scale Space Maxima 106-cell