1 Blob detection

Blob detection is a precursor used in a scale-invariant feature transform (SIFT), where the locations and strengths of interest points within an image are detected. If the strength of the interest point is scaled according to which frequency it corresponds to (i.e. lower frequencies result in larger blobs, higher frequencies with smaller blobs), then a blob-like feature can be extracted.

It should be noted that I am working individually, and I am competing for the speed/run-time competition.

2 Algorithm

MATLAB was used to implement the algorithm.

1. *Calculate Laplacian 'pyramid' of image*
   (a) Pyramid is in quotation marks above, because unlike a traditional pyramid that is downsampled at each layer, downsampling is an unnecessary step in the blob-detection algorithm.

   (b) The Gaussians are calculated using two functions, `gaus2dmesh()` and `gaus2dcrop()`. For rectangular images where the ratio between the maximum and minimum dimensions is greater than ∼4.75, `gaus2dmesh()` is used. Otherwise, `gaus2dcrop` is used. The value of ∼4.75 was calculated using by running run-time tests on both functions and comparing the results. A pattern was noticed, where this dimension ratio was the indicating factor for optimizing the calculation of the 2-dimensional Gaussian by selecting one method over another.

      i. `gaus2dcrop()` - Calculates the 1-dimensional Gaussian distribution given the variance and image dimensions. The maximum dimension is used for the length of the 1-dimensional Gaussian. This result is vector-multiplied with itself to create a matrix $\in \mathbb{R}^{dim_{max} \times dim_{max}}$. Then, the matrix is cropped to the size of the minimum dimension along the minimum dimension of the input image.

      ii. `gaus2dmesh()` - Calculates the 2-dimensional Gaussian distribution given the variance and image dimensions, using the commonly-used built-in MATLAB function `meshgrid()`

   (c) The Gaussian blurs are applied by convoluting the Gaussian with a greyscale version of the input image. This is more easily done in the frequency domain, so a 2-dimensional DFT is applied to each Gaussian as well as the greyscale input image. After element-wise multiplication, a 2-dimensional IDFT is used to return to the spatial domain.

   (d) Since a Laplacian can be approximated by taking the difference of two Gaussian distributions, then the original image is blurred using subsequently large Gaussian distributions and stored.

   It is necessary to note that this is only possible with a constant scaling of the standard deviation of the Gaussian. So,

$$\text{Laplacian} \approx G(k^{n+1}\sigma) - G(k^{n}\sigma),\ n \in [0, \text{number of layers} - 1]$$

   where k is traditionally approximately $\sqrt{2}$

The number of layers are calculated the following way

   i. Lower bound - An arbitrary minimum blob diameter is calculated, so that only $\sigma < \sigma_{min}$ remain in the potential standard deviations in the 'pyramid' calculation.
   ii. Upper bound - An arbitrary upper bound is calculated using a logarithmic relationship of the minimum dimension of the input image. This naturally results in a potential maximum blob-size that, if centered, will fit into the image along the smaller dimension.

(e) The difference of each adjacent Gaussian layer is what is used for the Laplacian approximation. This can be seen on the left side of Figure 1.

Each layer of the resulting Laplacian corresponds to 2-dimensional edge detection from a fine to coarse accuracy. If an blob-like feature is of the appropriate size, then the surrounding edges constructively build up to a peak, rather than simply magnifying the edge and falling into a depression toward the middle of the feature. Therefore, this peak (whether it be minimum or maximum) can be extracted and utilized as a SIFT interest point.

2. *Calculate the local extrema of the Laplacian*

(a) The traditional algorithm used to find local extrema would be to take the magnitude of each Laplacian layer and compare each pixel to its neighbors (extending the dimensions along the 'pyramid' layers, so one pixel would neighbor 26 others in a 3-dimensional space). If this pixel's magnitude is the largest, then it is a local extrema and saved as a SIFT interest point.

(b) The naïve implementation would be to sweep across each Laplacian layer with a $3 \times 3 \times 3$ cube, and determine whether the magnitude of the center is the maximum of the 27 points within the local group. However, this was too slow for my liking.

Instead, there was an effort made in finding a faster method. The idea of downsampling and upsampling using nearest-neighbor interpolation seemed very promising, as only 1 pixel (in possible 9 pixel positions of a 3×3 grid) would be used for comparisons of local maxima.

   i. A function called `sepblockfun()` (all credits to author are within sepblockfun.m) was utilized to split up an N-D array into the expected kernel size of 3×3 to perform localized operations (the operation implemented being maximum). However, this returned a downsized version of the input, which could not be compared to the upsampled image. This proved beneficial w.r.t computational efficiency, since there was no need to upsample the image in the first place.
   ii. A 4-dimensional array is preallocated and to fit the 2-dimensions of the Laplacians, the number of Laplacian layers, and each pixel for a 3×3 grid. For each pixel within the 3×3 grid, the local maximum is calculated using `sepblockfun()`
   iii. Then, 3 layers (selected along the layer dimension of the 4-dimensional array above) undergo a maximum operation to flatten into a resulting 3-dimensional array. The Laplacian is sampled periodically along each image dimension for all pixels $\in$ 9 possible pixel positions and compared directly to the maximum pixel value in the corresponding pixel-dimension of the 3-dimensional array. If the Laplacian pixel equals the localized maximum, then it is the localized maximum. Logical element-wise matrix multiplication is used to keep the maximum

magnitudes of the Laplacian for each Laplacian layer, while all other pixel values default to 0. This can be seen on the right side of Figure 1.

3. *Apply threshold to local extrema to calculate SIFT interest point*

   An arbitrary pixel magnitude threshold is selected to filter through the strongest, most prominent SIFT candidates. From experimentation, this value typically ranges from 10-50 in production of decent results. The lower the threshold, the more points are passed through, displaying more blobs. Subsequently, larger thresholds result in less blobs. The coordinates of each SIFT interest point are stored along the layer dimension.

4. *Display the image with blobs*

   For each layer, a quantized circle diameter is determined proportional to the standard deviation for that corresponding layer. Meaning, SIFT points in a layer are displayed with equal circle diameters regardless of their original magnitude. In MATLAB, the RGB input image is displayed and overlayed with circular figure objects.

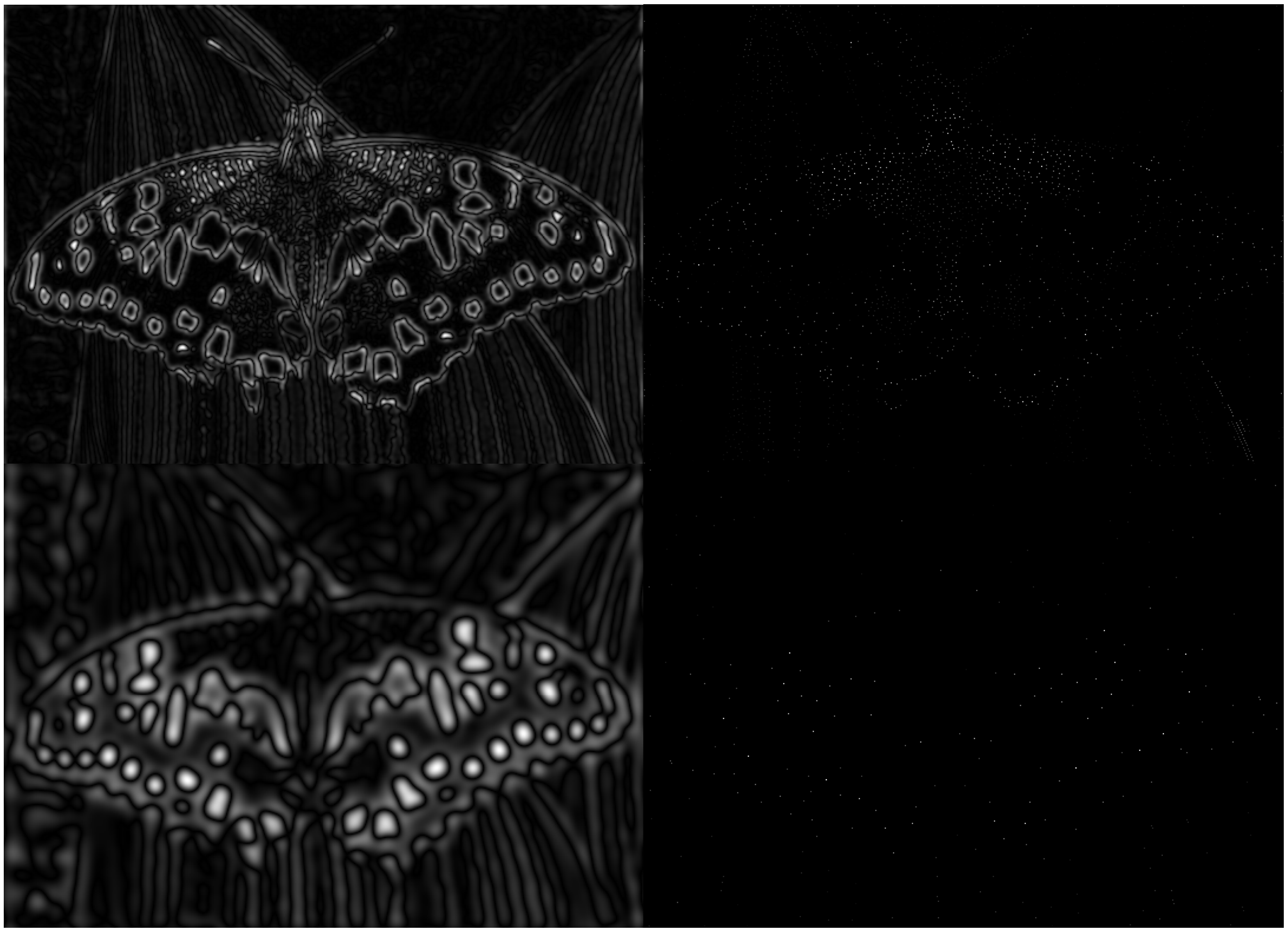


Figure 1: Laplacian on left, localized extrema on right. Layers 1 & 4. (Zoom in to see extrema)

3 Results

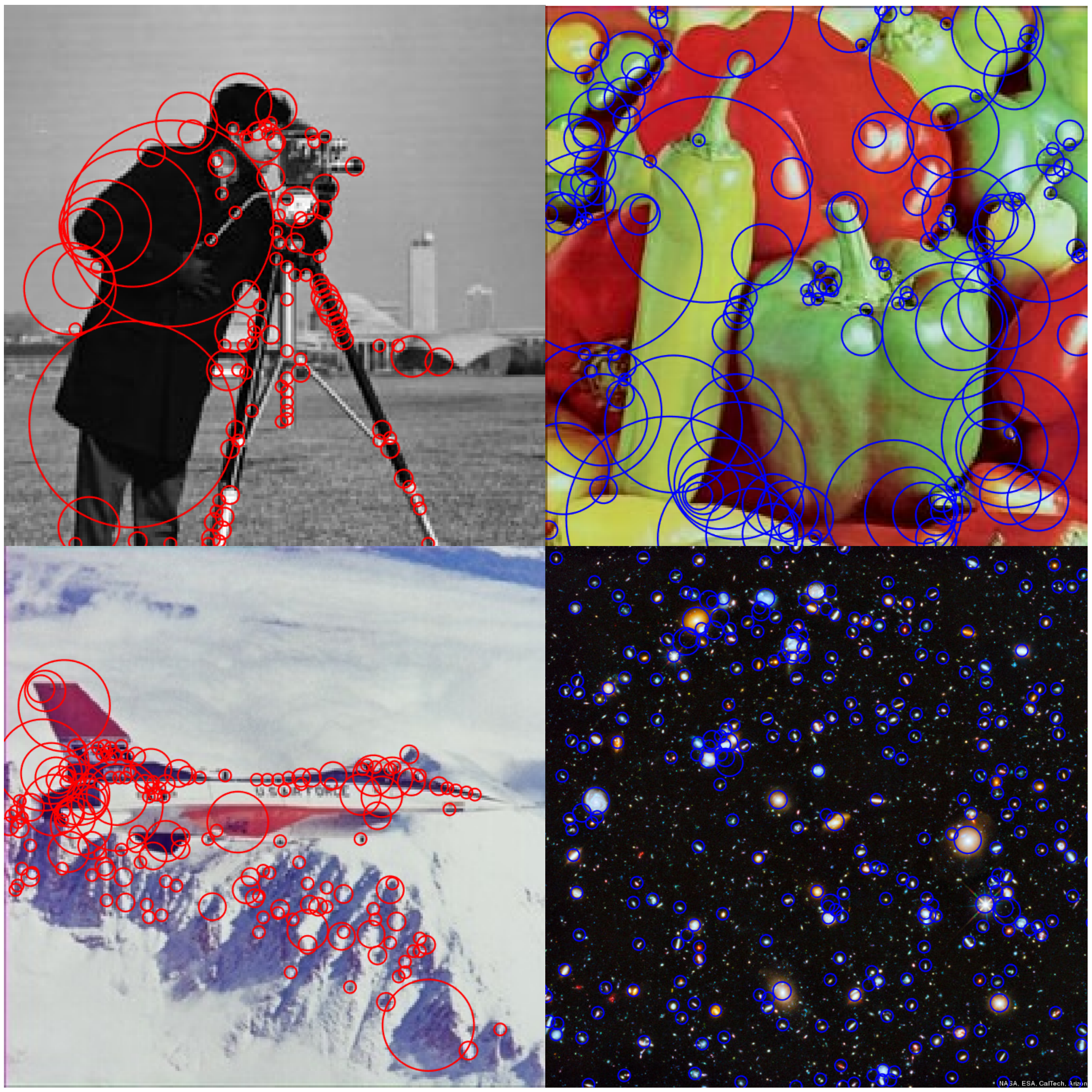


Figure 2: Some output images (cropped to square)

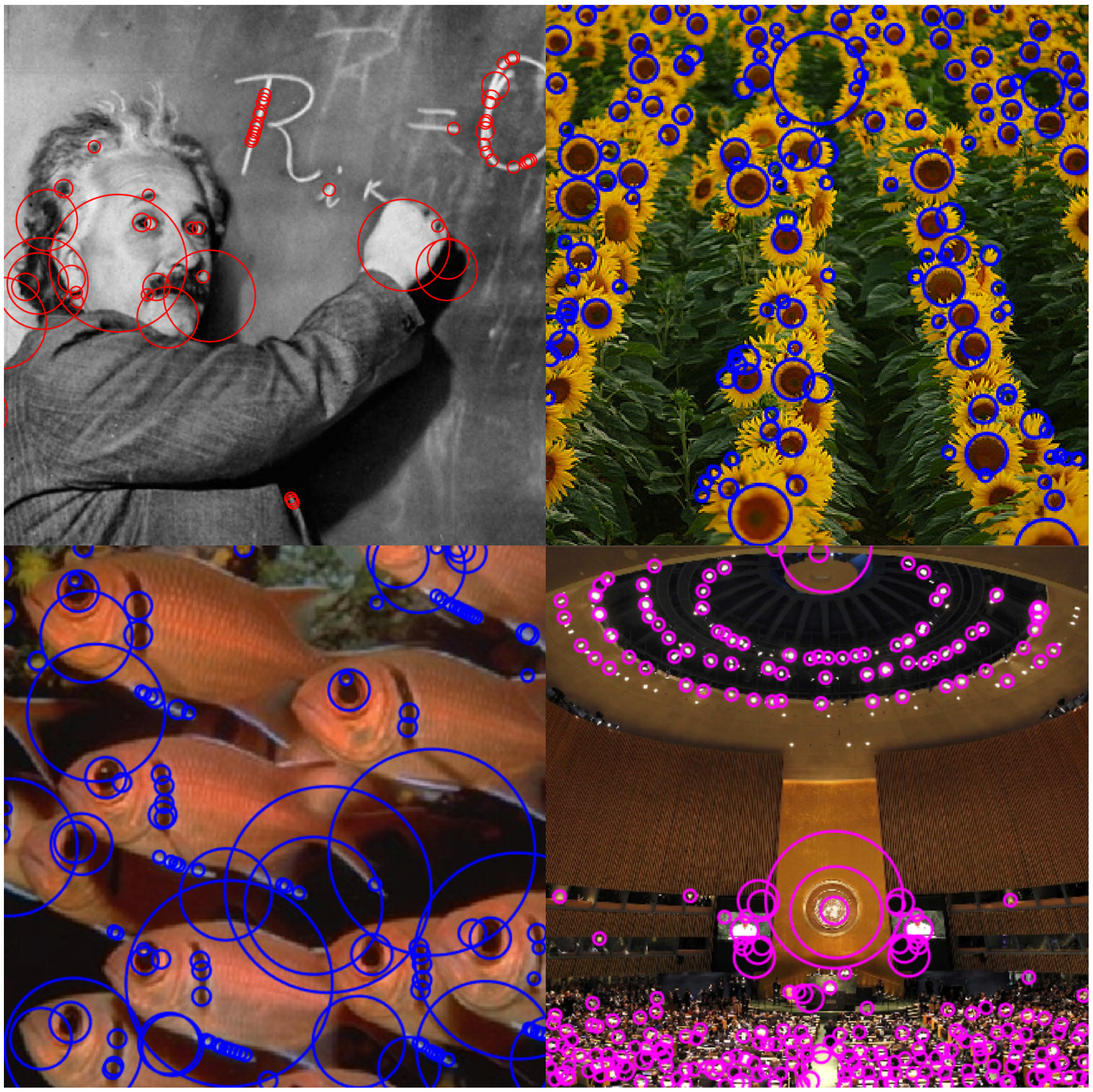


Figure 3: More output images (cropped to square)

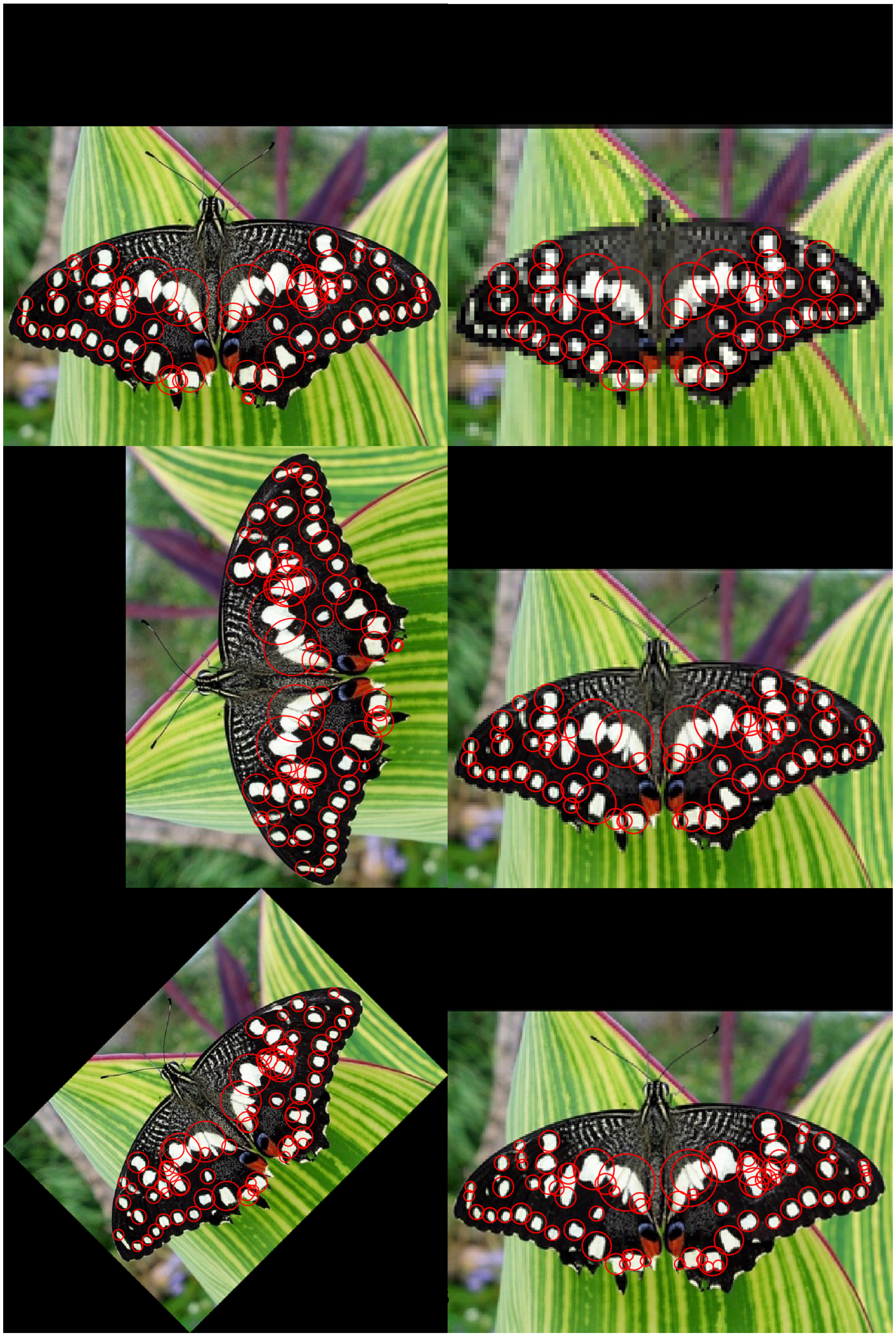


Figure 4: Showcasing scale and shift-invariant properties of blob detector

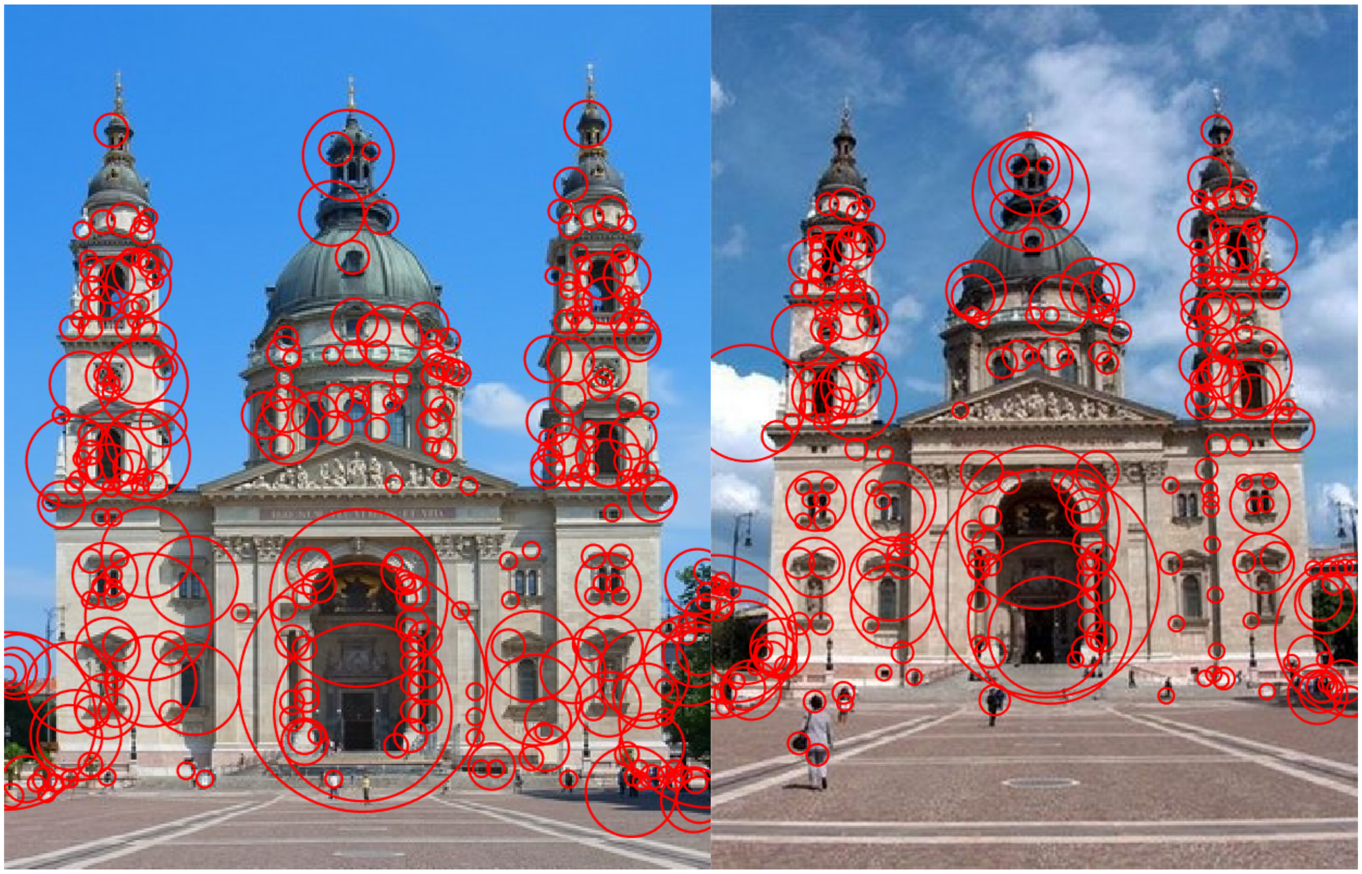
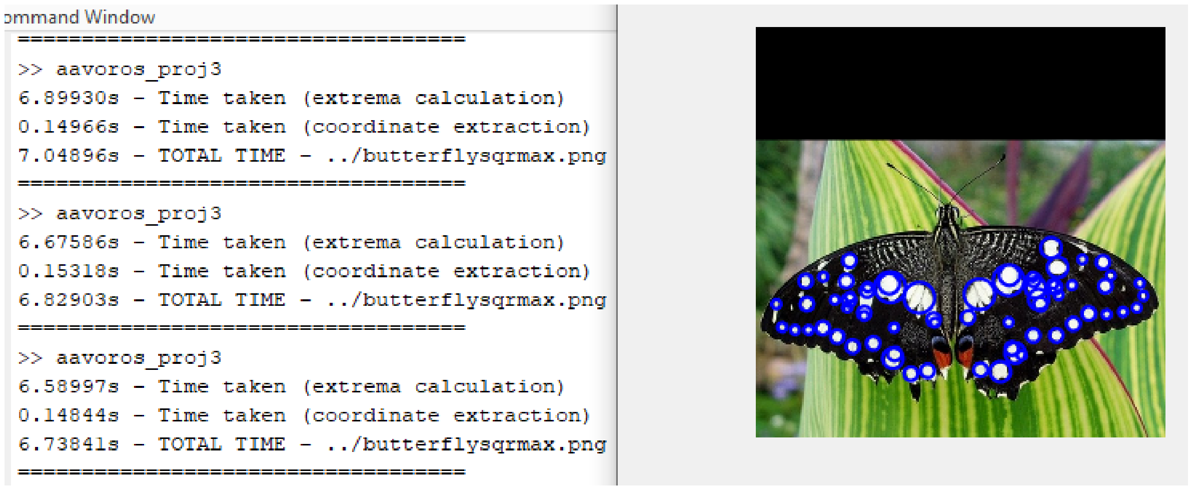


Figure 5: Precursor to SIFT image recognition, reverse transformation, image stitching, etc.



```
====================================
>> aavoros_proj3
6.89930s - Time taken (extrema calculation)
0.14966s - Time taken (coordinate extraction)
7.04896s - TOTAL TIME - ../butterflysqrmax.png
====================================
>> aavoros_proj3
6.67586s - Time taken (extrema calculation)
0.15318s - Time taken (coordinate extraction)
6.82903s - TOTAL TIME - ../butterflysqrmax.png
====================================
>> aavoros_proj3
6.58997s - Time taken (extrema calculation)
0.14844s - Time taken (coordinate extraction)
6.73841s - TOTAL TIME - ../butterflysqrmax.png
====================================
```

Figure 6: Run-time of large (2000 × 2000) image, showing output prompt.

3.1 Timing

Timing was performed using the built-in MATLAB functions `tic` and `toc`. Timing was performed around steps 1 through 3 in the Algorithm section. Time to display the image and blobs was NOT concluded, since this is purely dependent on how fast MATLAB can display a figure with other figure objects (the circles). Timing was only done on steps 1 through 3 because this is where all necessary calculations are made for the blob detection algorithm.

Many trials were run for each image. Some more than others, so this is meant to be a rough estimate to gauge the programs efficiency.

Image	Dimensions	Threshold	Average time (s)	Time per pixel (μs)
Figure 2	-	-	-	-
cameraman	200×200	22	0.075	1.875
peppers	256×256	17	0.150	2.289
airplane	256×256	20	0.150	2.289
space	821×900	12	1.990	2.693
Figure 3	-	-	-	-
einstein	480×640	21	0.632	2.057
sunflowers	357×328	20	0.295	2.519
fishes	335×500	15.5	0.418	2.496
politics	1214×1800	16	5.125	2.345
Figure 4	-	-	-	-
butterfly d100	100×100	32	0.022	2.200
butterfly d200	200×200	32	0.067	1.675
butterfly d300	300×300	33	0.165	1.833
butterfly vert	544×544	33	0.620	2.095
butterfly horz	544×544	33	0.620	2.095
butterfly angl	664×664	32	1.220	2.767
Figure 5	-	-	-	-
bazilika1	416×340	16	0.350	2.475
bazilika2	600×450	18	0.570	2.111
Figure 6	-	-	-	-
butterfly d2000	2000×2000	40	6.900	1.725

Table 1: Timing results of the figures above

The tests were run on MATLAB 2021b, 64-bit Windows 10, Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz 2.71 GHz, 8.00 GB (7.86 GB usable) RAM

4 Instructions to run

1. Open `aavoros_project03.m` in MATLAB
2. **Before running**
   (a) Change the `FILEPATH` variable to the path of the input image on Line 2.
   (b) (Optional) - Change the `layer_scale` variable on Line 13. This is roughly $\sqrt{2}$, so can be anything from 1.3 - 1.6, but should typically be unchanged
   (c) Change the `THRESHOLD` variable on Line 21 to determine what threshold to filter SIFT candidate points.
   (d) Change the `LINECOLOR` and `LINEWIDTH` variables on Lines 27 & 28. All possible colors can be seen in the `rectangle` function documentation.
3. **Running**
   (a) Press the 'Run' button to run all
   (b) If a different `THRESHOLD`, `LINECOLOR`, or `LINEWIDTH` need to be specified, only 'Run Section' on the coordinate extraction & display section of the script (Line 18 onward)