ENGGEN 131

Introduction to Engineering Computation
and Software Development

MATLAB

Image Fingerprinting

Semester 2, 2023

The Department of Engineering Science
The University of Auckland

Academic Integrity

Write your own code, and **do not give your code to anyone else**. This project needs to be your work, and you should create your own code without using generative AI tools such as ChatGPT. It is an individual project, not a group project. Please note we use software that detects plagiarism on every project submission, so we will catch you if you copy.

If a student copies work from another student; online tools; or through other means that violate academic integrity; at a minimum, all students involved will receive a mark of zero for that assessment. Subsequently, the students will also have their names recorded in the university's Register of Deliberate Academic Misconduct.

If you have already been found guilty of academic misconduct on another course or assignment, the penalty will likely be even more significant (e.g., a fine and a failing grade). If you have any queries about what is or is not academic misconduct, please ask so that we can ensure everyone understands what is acceptable behaviour and what is not.

Getting Started

We have broken down the project into several manageable pieces by dividing the logic into separate functions. These functions vary in difficulty, but the project specification thoroughly explains what we expect each function to do. You can complete some functions independently, while for other functions, you will likely find it helpful to call previously completed functions (but they can still be completed independently).

You are also free to write any additional helper functions that you find useful. If you do, submit any helper functions to MATLAB Grader underneath the function that is calling it.

Your goal should be to complete as many of these functions as possible.

Begin by reading through the entire project specification to get a sense of the project as a whole. Then it would be good to collect your thoughts and devise a plan of attack. For example:

- Are you going to attempt the "easy" functions first?
- Are you going to attempt the functions in order?
- Are you going to create test scripts before writing functions?
- Are you going to finish one function a day? Or all of them in one sitting?

Testing

Testing your code before submitting it to MATLAB Grader is always a good idea. Remember that MATLAB Grader is not a development environment; you should only use it when you are ready to submit your functions.

We provide a few examples that you can use to test your code, but these are by no means exhaustive. Please be aware that these will not test the full functionality of your functions. We expect you to write your own tests and use various inputs (such as edge cases) to test your functions.

MATLAB Functions

The lectures taught us most of the functions required to complete this project. However, exploring unfamiliar functions is also a skill to improve. Some other functions are available to use but may require us to do a little bit of research. Recall that we can use the MATLAB Help system or the MATLAB Documentation, books, search engines or videos when learning these new ideas.

We can use any functions covered in lectures and most other functions that are part of MATLAB's core distribution (with some minor exclusions clarified further in the project specification).

For this project, you may not use functions only available in toolboxes.

MATLAB features a large number of optional toolboxes that contain extra functions. The purpose of this project is for you to get practice at coding, not to call some toolbox functions that do all the hard work for you.

If you are uncertain whether a particular function is part of MATLAB's core distribution or not, you can use the `which` function as follows:

```
>> which("function")
```

where *`function`* is replaced by the name of the function we are interested in.

Check the text output to see if the directory directly after the word `toolbox` is `matlab` (which means it is core) or something else (which means it is from a toolbox). For example:

```
>> which("help")
C:\Program Files\MATLAB\R2023a\toolbox\matlab\helptools\help.m
```

Some built-in functions may have output with prefix `built-in`, which confirms it is a built-in function. For example:

```
>> which("disp")
built-in (C:\Program Files\MATLAB\R2023a\toolbox\matlab\lang\disp)
```

Other functions, like the following, do not belong to MATLAB's core distribution for example:

```
>> which("normcdf")
C:\Program Files\MATLAB\R2023a\toolbox\stats\stats\normcdf.m
```

The output shows us that the `normcdf` function is part of the Statistics and Machine Learning Toolbox; therefore, you cannot use this function for the project.

Submission and Marking

You will submit all functions to MATLAB Grader for marking.

You can find more details on how we mark this project in the Marking section at the end of the specification.

Image Fingerprinting

The Image Fingerprinting project will involve writing MATLAB code that will "hash" images to find their image fingerprint. Hashing is taking information, such as a string of text or an image and transforming it into a typically shorter fixed-length value or key that is easier to work with. In our case, the hashes we use will create image fingerprints we can use to compare images and determine their similarity efficiently.

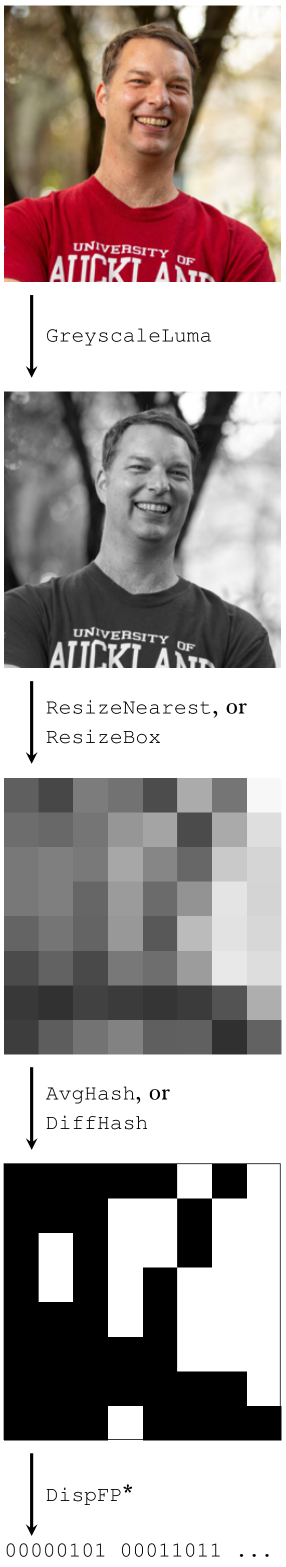


Figure 1: The `ImageFingerprint` function (plus the `DispFP` function).

Image fingerprinting can be used in various interesting ways, for example, identifying copyright issues with images or searching a system for an original image when we only have the thumbnail-sized version. For this project, we will implement several functions that allow us to create a digital fingerprint of an image and use that to find similar images. Our functions are easier to implement than other approaches, which means our results or efficiency might not be as perfect as some other software but will hopefully give a good result with the right set of images.

To create our image fingerprint, we need to reduce the amount of data used to represent the image while also maintaining the core aspects of the image that make it identifiable (`ImageFingerprint`). Ideally, these features resist change (i.e., if we edit the image is slightly, its fingerprint should remain unchanged). We cannot to do this perfectly, but we will find it works for some small image manipulations. These steps are visualised in Figure 1 and include:

1. Strip the image of its colour data by converting it to a Greyscale image (`GreyscaleLuma`), which reduces our image to a third of its size (i.e., from 3 layers of colour to 1 layer of grey values).
2. Reduce the size of the image (`ResizeBox` and `ResizeNearest`) to an 8-by-8 array (for `AvgHash`) or 8-by-9 array (for `DiffHash`), significantly reducing the data amount. Surprisingly, this still maintains enough information to distinguish between images.
3. Convert this small array of `uint8` values into a single 64 bit hash of `1`s and `0`s. While for illustrative purposes we show the hash in Figure 1 as a 2D grid of black and white squares, for simplicity we will store these 64 bits in a 1-by-64 `logical` row vector. We will do this conversion with two different hashing algorithms, Average Hashing (`AvgHash`) and Differential Hashing (`DiffHash`).

With an image's 64-bit image fingerprint, we can efficiently compare it to other images which have had their fingerprint identified (`FingerprintCollection`). We can visually check the bits against each other manually (`DispFP`). But it is more efficient to automatically determine how many of them differ in each fingerprint (`HammingDistance`). If we have a low distance, the images are likely very similar! Once all the comparisons are made, we output our results in the order of most to least similar (`RankSimilarity`).

We will provide sample images and the `ImageSearch` script visualised in Figure 2, which calls the `FingerprintCollection` and `RankSimilarity` functions to find images most similar to a specified image.

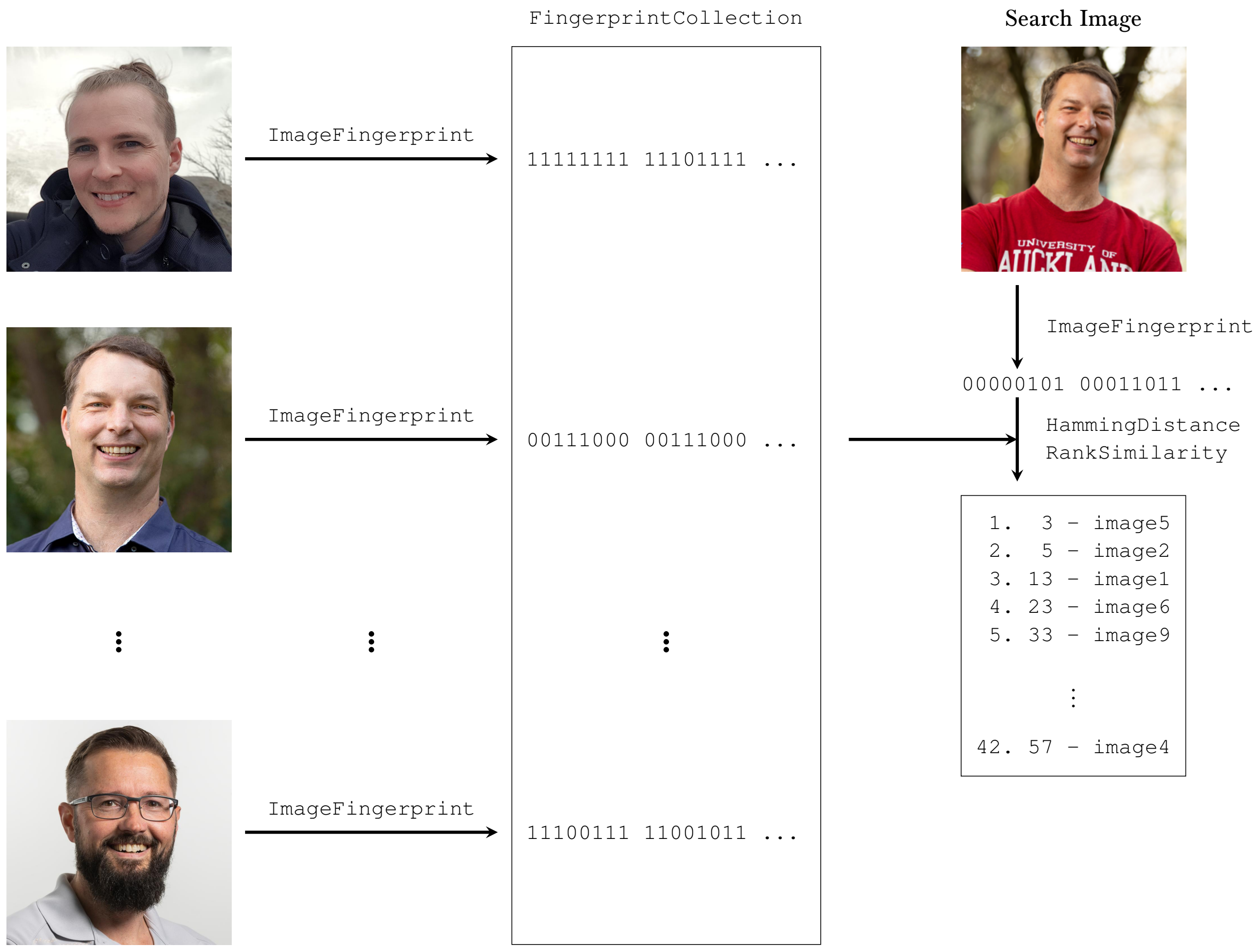


Figure 2: The `ImageSearch` script.

Function Overview

Function	Difficulty Level*	Type
`GreyscaleLuma`	Easy	Image processing.
`ResizeNearest`	Medium	Image processing.
`AvgHash`	Easy	Hash algorithm.
`DispFP`	Easy	Text processing.
`HammingDistance`	Easy	Array processing.
`DiffHash`	Medium	Hash algorithm.
`ResizeBox`	Hard	Image processing.
`ImageFingerprint`	Easy	Comprehensive function.
`FingerprintCollection`	Easy	Comprehensive function.
`RankSimilarity`	Medium	Text formatting.

*The difficulty level is our approximation. You may find easy tasks hard, and you may find hard tasks easy. Remember, if you are finding something difficult, take a step back, try rereading the project specification, work through the six steps of problem-solving or test your code more while stepping through with the debugger.

Keep the following general principles in mind:

- Function names are case-sensitive. The function we write must match **exactly** those in the project specification.
- The order of the function input and output arguments matter.
- The data type of the function input and output arguments matter.
- The shape of the function input and output arguments matter.
- Test the function ourselves **before** submitting it to MATLAB Grader.

GreyscaleLuma

The `GreyscaleLuma` function converts the supplied RGB colour image to a greyscale version by calculating a weighted sum for each pixel.

Input:

- An *m*-by-*n*-by-3 `uint8` array representing an RGB colour image.

Output:

- An *m*-by-*n*-by-1 `uint8` array representing a greyscale image.

Suppose we have a small colour image stored in the 3D `uint8` array named `small_image`. Layers 1, 2, and 3 represent the red r, green g, and blue b pixel intensities, respectively.

To convert the image from colour to greyscale, we would use the `GreyscaleLuma` function to return a greyscale image (as the 2D `uint8` array `greyscale_image`) as follows:

```
>> greyscale_image = GreyscaleLuma(small_image)
```

The `GreyscaleLuma` function will need to calculate a grey pixel value p_g for each pixel in the colour image, using the corresponding (r, g, b) value of each pixel and applying the following weighted sum formula [1]:

$$p_g = 0.2126r + 0.7152g + 0.0722b$$

For example, at pixel position $(1, 1)$, we might have $(r, g, b) = (200, 55, 80)$. Solving our weighted formula gives:

$$\begin{aligned} p_g^{(1,1)} &= 0.2126 \times 200 + 0.7152 \times 55 + 0.0722 \times 80 \\ &= 87.6320 \end{aligned}$$

So pixel $(1, 1)$ has the value `88` in the output greyscale image.

Note: We must round the pixel value to the nearest integer.

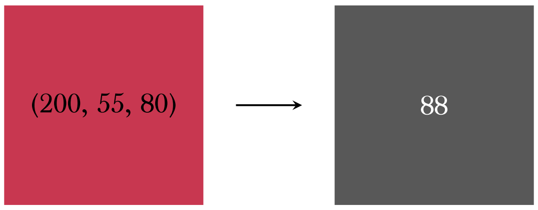


Figure 3: Pixel with value $(r, g, b) = (200, 55, 80)$ converted to grey with the `GreyscaleLuma` function.

Careful! Output Data Type

The output returned by `GreyscaleLuma` should be of the `uint8` data type. Take care when doing the calculations that you are using an appropriate type for the intermediate calculations (which will involve non-integer values).

Stop: Forbidden Functions

We may **not** use the built-in `rgb2gray`, `im2gray` functions (or any other similar MATLAB function) that automatically convert a colour image to greyscale to complete this task!

[1] There are several commonly used weightings for converting colour images to greyscale. We are using one referred to as Luma.

ResizeNearest

The `ResizeNearest` function uses the Nearest Neighbour Interpolation algorithm to resize an image to the specified dimensions.

Inputs:

- An m-by-n-by-p `uint8` array representing a greyscale ($p = 1$) or a RGB colour ($p = 3$) image.
- A 1-by-2 `double` array containing two values (i, j) representing the desired height i (i.e., i rows) and width j (i.e., j columns) to resize the image to.

Output:

- An i-by-j-by-p `uint8` array representing the resized greyscale ($p = 1$) or RGB colour ($p = 3$) image.

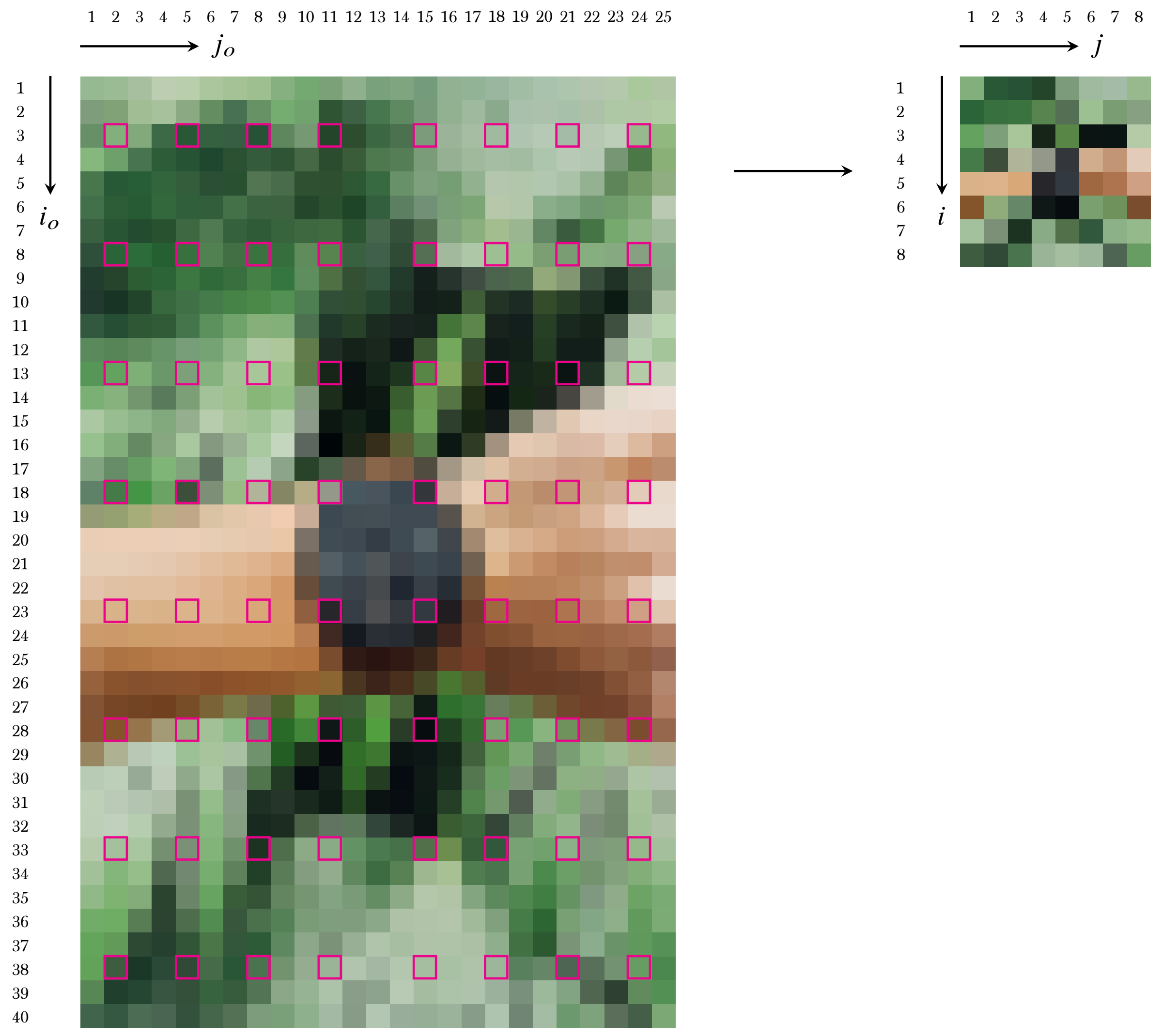


Figure 4: Visualisation of the Nearest Neighbour Interpolation algorithm.

Nearest-Neighbour Interpolation is an algorithm that resizes an image by selecting pixel values nearest to specific pixels in the original image. It calculates the new pixel's relative position in the original image and takes the pixel value nearest to the calculated relative position.

To start using this algorithm, we need to know the row and column size ratios (R_{Row} and R_{Col}) between the input and output image:

$$R_{\text{Row}} = \frac{\text{Rows in Original Image}}{\text{Rows in Resized Image}} \qquad R_{\text{Col}} = \frac{\text{Columns in Original Image}}{\text{Columns in Resized Image}}$$

We can then use the above ratios to calculate the relative position (i_o, j_o) of our resized pixel (i, j) in the original image with:

$$i_o = (i - 0.5) \times R_{\text{Row}} \qquad\qquad j_o = (j - 0.5) \times R_{\text{Col}}$$

Note: We subtract 0.5 as the pixel has a "width" of 1, so we must use its centre point when calculating its relative position in the original image.

Note: After calculating values for the pixel relative positions (i_o, j_o), we need to round our positions **up** to the nearest integer (as (i_o, j_o) values in the interval $[0, 1]$ would represent pixel 1, interval $[1, 2]$ would represent pixel 2 etc.).

Finally, we copy the pixel values at the relative pixel positions (i_o, j_o) back to our resized image positions (i, j).

For example, consider resizing the 40-by-25 image to an 8-by-8 image shown in Figure 4. The row and column size ratios (R_{Row} and R_{Col}) are:

$$R_{\text{Row}} = \frac{40}{8} = 5 \qquad\qquad R_{\text{Col}} = \frac{25}{8} = 3.125$$

Thus, for pixel $(i, j) = (3, 4)$ in our resized image, the pixel relative position (i_o, j_o) is calculated as:

$$i_o = (3 - 0.5) \times 5 \qquad\qquad j_o = (4 - 0.5) \times 3.125$$
$$i_o = 12.5 \qquad\qquad j_o = 10.9375$$

After rounding up, we obtain the relative position $(i_o, j_o) = (13, 11)$. Therefore, pixel $(3, 4)$ is assigned the same value as pixel $(13, 11)$ from the original image.

Image Fingerprinting

At a minimum, `ResizeNearest` must be able to scale down the size of a greyscale image for our Image Fingerprinting to work.

However, if we correctly implement `ResizeNearest`, it should be able to scale an RGB colour image (3D array) **and** a greyscale image (2D array) to any size (larger **and** smaller).

Careful!

Implementations of the nearest neighbour interpolation algorithms can have slight variations. Make sure the `ResizeNearest` function implements it as the brief has described **exactly**. Pay particular attention to the rounding and which pixels in the output image correspond to which pixels in the input image.

Stop: Forbidden Functions

You may **not** use the built-in `imresize` function (or any similar MATLAB function) that automatically resizes an image to complete this task!

AvgHash

The `AvgHash` function uses the Average Hashing algorithm to convert an m-by-n greyscale image into a hash containing mn bits.

Input:

- An m-by-n `uint8` array representing a greyscale image.

Output:

- A 1-by-mn `logical` row vector representing the average hash of the greyscale image.

Average Hashing is an algorithm that reduces fingerprint size by comparing each pixel value to the average pixel value of the greyscale image.

The first step in the algorithm is to find the average value of the pixels in the image array rounded to the nearest integer. Next, we iterate through each pixel in the image, starting at index $(1, 1)$ and proceeding column by column, then row by row (i.e., left to right, top to bottom), and comparing the pixel value to the average pixel value.

- If the pixel value is greater than or equal to the average value, it should be represented by the `logical 1` (or `true`) in the output array.
- If the pixel value is less than the average, it should be represented by the `logical 0` (or `false`) in the output array.

After comparing each pixel in the array, we should have created a `logical` row vector.

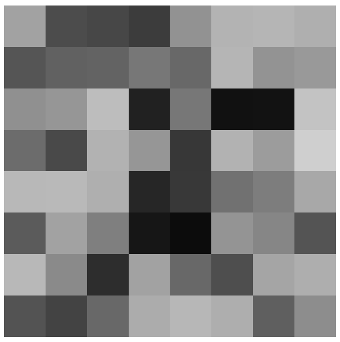


Figure 5: Greyscale image represented by the image array `watch_nearest_grey`.

For example, consider the following 8-by-8 image array in Figure 5 from resizing the 40-by-25 image in Figure 4 and converting to greyscale with the `GreyscaleLuma` function:

```
>> watch_nearest_grey

watch_nearest_grey =

  8×8 uint8 matrix

   162    76    71    60   146   179   181   175
    85    97    99   119   104   181   147   153
   144   150   189    33   119    17    18   195
   108    73   178   150    55   178   156   207
   184   185   175    38    56   113   125   168
    91   162   127    22    12   148   134    84
   184   138    45   162   104    78   165   174
    83    67   104   172   183   174    95   141
```

The average pixel value is 123.4062, which rounds to 123. After comparing each pixel in the array to the average value, we should have created a 1-by-64 `logical` row vector, the 64-bit image fingerprint. In particular, we will obtain the following fingerprint for `watch_nearest_grey`:

```
>> fingerprint = AvgHash(watch_nearest_grey)
fingerprint =
  1×64 logical array
  Columns 1 through 19
   1   0   0   0   1   1   1   1   0   0   0   0   0   1   1   1   1   1   1
  Columns 20 through 38
   0   0   0   0   1   0   0   1   1   0   1   1   1   1   1   1   0   0   0
  Columns 39 through 57
   1   1   0   1   1   0   0   1   1   0   1   1   0   1   0   0   1   1   0
  Columns 58 through 64
   0   0   1   1   1   0   1
```

If we use our `DispFP` function, we can tidy the above output to be the following:

```
>> fingerprint = AvgHash(watch_nearest_grey);
>> DispFP(fingerprint);
10001111 00000111 11100001 00110111 11100011 01100110 11010011 00011101
```

Image Fingerprinting

At a minimum, `AvgHash` must be able to create a fingerprint from an 8-by-8 greyscale image for our Image Fingerprinting to work.

However, if we correctly implement `AvgHash`, it should be able to create a fingerprint from any *m*-by-*n* greyscale image.

Careful! Output Data Type

The `AvgHash` function should return a `logical` row vector (i.e., each element value is a `logical 1` or `0` (i.e., `true` or `false`)).

A `logical` array differs from a `double` array that can contain large numbers (or the `char` vector displayed above).

You can check the data type using the `class` function or by looking at the Workspace.

Tip: Start Small

Start testing your code with suitably small arrays like the following **before** proceeding to larger arrays:

```
>> tinyGrey = [10 20 30; 40 50 60; 70 80 90];
>> a = AvgHash(tinyGrey)
```

DispFP

The `DispFP` function displays the image fingerprint in a pretty format on one line.

Input:

- A 1-by-n `logical` row vector representing an image fingerprint.

Output:

- None.

By default, when MATLAB displays arrays, it will scale the columns of values displayed per line by the window width. MATLAB will also continue to display the column's row values until all the columns are displayed. This behaviour could be more friendly when we want to view our fingerprints quickly. So `DispFP` is a helper function we will write to enable us to view the entire fingerprint with one line in the Command Window.

The `DispFP` function must display the fingerprint (as `1`s and `0`s) on a single line with a space character every 8 bits of the fingerprint and end with a new line. We might call `DispFP` as follows with the output of the `AvgHash` function:

```
>> fingerprint1 = AvgHash(watch_nearest_grey);
>> DispFP(fingerprint1);
10001111 00000111 11100001 00110111 11100011 01100110 11010011 00011101
```

If the length of the input n is not a multiple of 8, then the last set of `1`s and `0`s will not be in a group of 8. For example:

```
>> DispFP(logical([1 1 1 0 0 0 1 1 1 0 0 0])); % twelve elements
11100011 1000
```

Calling `DispFP` consecutively with different fingerprints allows us to identify similarities and differences visually. For example:

```
>> fingerprint1 = AvgHash(watch_nearest_grey);
>> fingerprint2 = DiffHash(watch_box_grey);
>> DispFP(fingerprint1); DispFP(fingerprint2);
10001111 00000111 11100001 00110111 11100011 01100110 11010011 00011101
11000001 00110010 00110100 00110010 11110000 01101010 11010100 10000111
```

Note: The `DispFP` function should not display whitespace before or after our fingerprint portion (aside from the new line character).

Careful! Function Output

The `DispFP` function should not return any output.

However, it has an "output", the fingerprint displayed in the Command Window.

HammingDistance

The `HammingDistance` function compares two image fingerprints, f_1 and f_2, and determines the number of positions at which the corresponding values differ. The number of positions that differ is known as the Hamming distance. A small Hamming distance indicates similar images.

Inputs:

- A 1-by-n `logical` row vector for fingerprint f_1.
- A 1-by-n `logical` row vector for fingerprint f_2.

Output:

- A integer representing the number of differing bits between fingerprints f_1 and f_2.

Note: Assume the two input hashes will always have the same length n.

Consider the following example that visualises the differing bits in bold:

```
>> fp1 =

  1×8 logical array

   1   0   1   1   1   0   1   0
```

```
>> fp2 =

  1×8 logical array

   1   1   1   1   0   0   1   0
```

Alternatively, consider the same fingerprints visualised with the `DispFP` function (bold added for emphasis):

```
>> DispFP(fp1); DispFP(fp2);
10111010
11110010
```

The bits are the same except at indices 2 and 5, so we would return a distance value of $1 + 1 = 2$. Therefore, we would expect the following output with `HammingDistance`:

```
>> ham = HammingDistance(fp1, fp2)

ham =

     2
```

Image Fingerprinting

At a minimum, `HammingDistance` must find the distance for fingerprints containing 64 bits (i.e., 1-by-64 `logical` row vectors) for our Image Fingerprinting to work.

However, if we correctly implement `HammingDistance`, it should be able to determine the distance between two fingerprints of any equal length n.

DiffHash

The `DiffHash` function converts an m-by-$(n+1)$ greyscale image into a hash containing mn bits using the Differential Hashing algorithm.

Input:

- An m-by-$(n+1)$ `uint8` array representing a greyscale image.

Output:

- A 1-by-mn `logical` row vector representing the differential hash of the greyscale image.

Note: Assume $n > 0$.

Differential Hashing is an algorithm that reduces fingerprint size by comparing each pixel value to the pixel value to the right of itself in the greyscale image.

It follows the same approach as the Average Hash algorithm. However, instead of comparing against the average value of the array, we compare a pixel value with the pixel to the right of that pixel. To obtain mn bits, we need m rows and $n+1$ columns (rather than n) since we are comparing differences.

We iterate through each pixel in the image, starting at index $(1,1)$ and proceeding column by column, then row by row (i.e., left to right, top to bottom), and comparing the current pixel value to the pixel one column to the right.

- If the pixel value is greater than or equal to the pixel value one column to the right, it should be represented by the `logical 1` (or `true`) in the output array.
- If the pixel value is less than the pixel one column to the right, it should be represented by the `logical 0` (or `false`) in the output array.
- If we are at the end of a row (i.e. the last column in the row), we cannot compare it to a pixel to the right, as there is not one (which is why we need $n+1$ pixels in each row to get n comparisons).

After comparing each pixel in the array, we should have created a `logical` row vector.

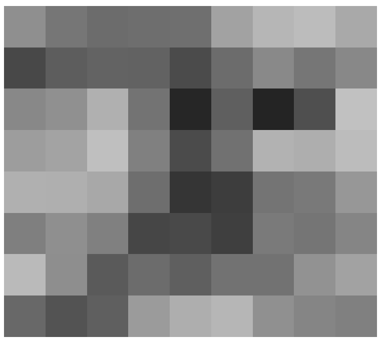


Figure 6: Greyscale image represented by the image array `watch_box_grey`.

For example, consider the following 8-by-9 image array in Figure 6 obtained from resizing the 40-by-25 image in Figure 7 using the `ResizeBox` function and converting to greyscale with the `GreyscaleLuma` function:

```
>> watch_box_grey

watch_box_grey =

  8×9 uint8 matrix

   143   118   108   110   111   162   182   188   169
    72    93    99    98    75   108   137   118   136
   136   144   176   115    38    95    36    79   193
   157   163   191   128    75   113   178   174   188
   176   175   168   110    53    61   116   121   151
   127   143   128    70    73    63   122   117   133
   186   142    90   108    95   114   114   146   162
   104    83    95   155   174   182   144   133   128

>> fingerprint = DiffHash(watch_box_grey)

fingerprint =

  1×64 logical array

  Columns 1 through 19

   1   1   0   0   0   0   0   1   0   0   1   1   0   0   1   0   0   0   1

  Columns 20 through 38

   1   0   1   0   0   0   0   1   1   0   0   1   0   1   1   1   1   0   0

  Columns 39 through 57

   0   0   0   1   1   0   1   0   1   0   1   1   0   1   0   1   0   0   1

  Columns 58 through 64

   0   0   0   0   1   1   1
```

If we use our `DispFP` function, we can tidy the above output to be the following:

```
>> fingerprint = DiffHash(watch_box_grey);
>> DispFP(fingerprint);
11000001 00110010 00110100 00110010 11110000 01101010 11010100 10000111
```

Image Fingerprinting

At a minimum, `DiffHash` must be able to create a fingerprint from an 8-by-9 greyscale image for our Image Fingerprinting to work.

However, if we correctly implement `DiffHash`, it should be able to create a fingerprint from any m-by-$(n+1)$ greyscale image.

ResizeBox

The `ResizeBox` function uses the Box Sampling algorithm to resize an image to the specified dimensions.

Inputs:

- An m-by-n-by-p `uint8` array representing a greyscale ($p = 1$) or a RGB colour ($p = 3$) image.
- A 1-by-2 `double` array containing two values (i, j) representing the desired height i (i.e., i rows) and width j (i.e., j columns) to resize the image to.

Output:

- An i-by-j-by-p `uint8` array representing the resized greyscale ($p = 1$) or RGB colour ($p = 3$) image.

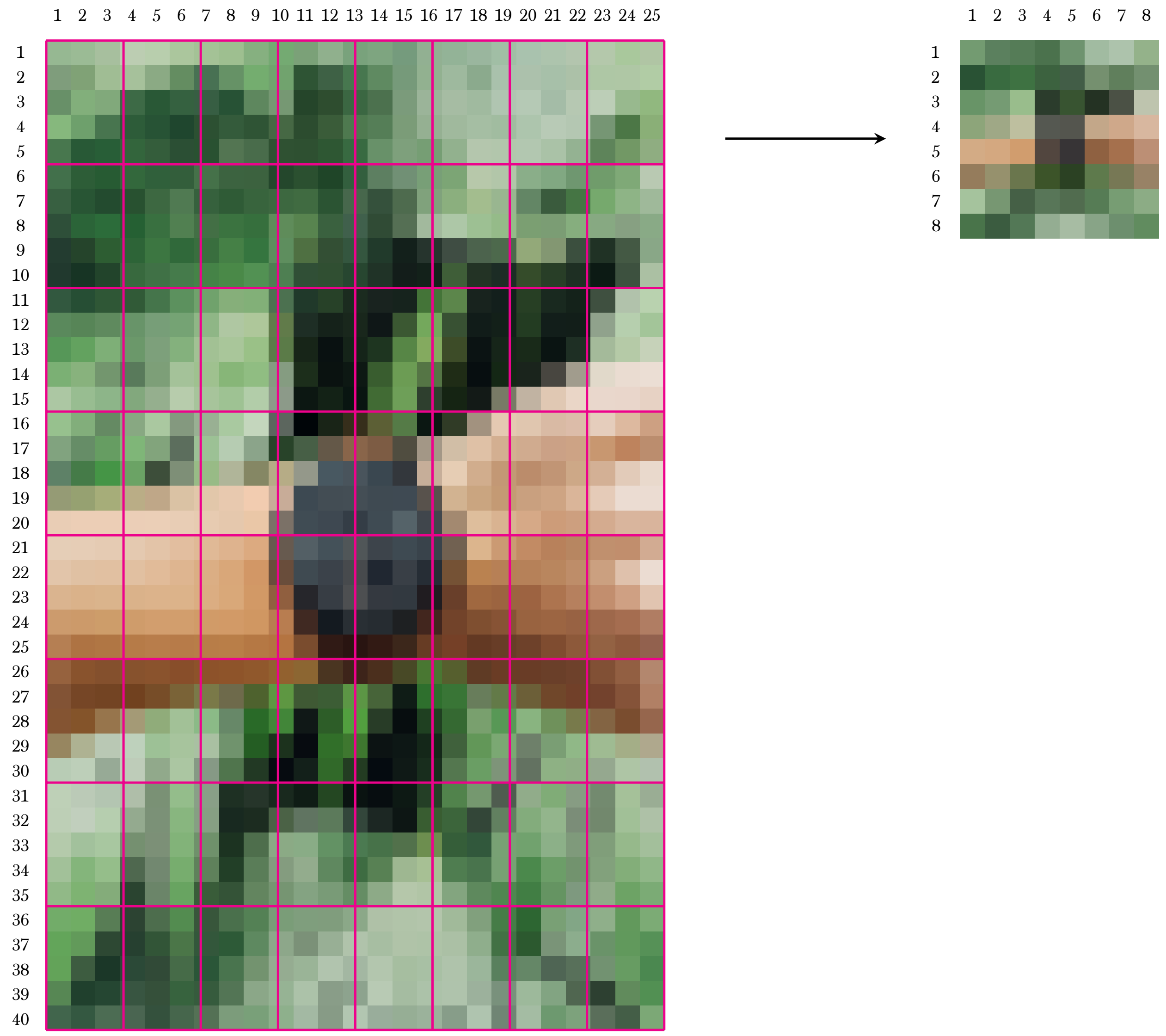


Figure 7: Visualisation of the Box Sampling algorithm.

One of the problems with the Nearest Neighbour Interpolation algorithm is it ignores many pixel values when resizing an image, which can lead to poor results. One way to use all the pixel values is through the Box Sampling algorithm.

Box Sampling is an algorithm that resizes an image by calculating the average pixel value in a group of pixels in the original image. It calculates bounding boxes that divide the original image into the same number of boxes as pixels in the resized image. Then, it calculates the average pixel value in the box to determine the pixel value of the resized image.

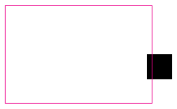
For a set of RGB pixels, we calculate the average RGB pixel value by finding the average of each layer of colour separately (i.e., the average red pixel value is found by averaging all the red colour intensities in the set etc.).

Note: We must round the pixel values to the nearest integer.

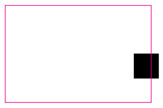
Consider resizing the 40-by-25 image to an 8-by-8 image shown in Figure 7. Using the Box Sampling algorithm, we would divide the image into an 8-by-8 grid and then find the resized pixel value by averaging the colour values of each of the boxes. However, some pixels overlap multiple bounding boxes. To deal with pixels that fall on the boundary of a box, we will use the following rule: The pixel is included in the average calculation for a box if **both** of the following conditions are true:

1. Half or more of the pixel width falls within the box.
2. Half or more of the pixel height falls within the box.

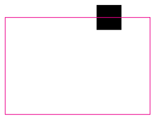
This rule approximates to only including pixels for which most of the pixel is inside the box. Consider the following rule examples:



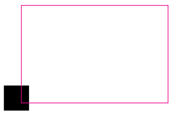
Less than half of the pixel's width and all of the pixel's height is inside the bounding box. Therefore, we **exclude** the pixel from the average calculation.



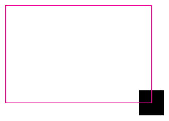
More than half of the pixel's width and all of the pixel's height is inside the bounding box. Therefore, we **include** the pixel in the average calculation.



All of the pixel's width and exactly half of the pixel's height is inside the bounding box. Therefore, we **include** the pixel in the average calculation.



Less than half of the pixel's width and more than half of the pixel's height is inside the bounding box. Therefore, we **exclude** the pixel from the average calculation.



Half of the pixel's width and half of the pixel's height is inside the bounding box. Therefore, we **include** the pixel in the average calculation.

The last example illustrates that in rare cases, pixels that intersect with a corner may be included in the average calculation, even though most of the pixel does not fall inside the box. Similarly, pixels may be included in multiple boxes.

Image Fingerprinting

At a minimum, `ResizeBox` must be able to scale down the size of a greyscale image for our Image Fingerprinting to work.

However, if we correctly implement `ResizeBox`, it should be able to scale down an RGB colour image (3D array) **and** a greyscale image (2D array).

Note: The Box Sampling algorithm only works for scaling down.

Stop: Forbidden Functions

You may **not** use the built-in `imresize` function (or any similar MATLAB function) that automatically resizes an image to complete this task!

Less guidance is given for this task. You are expected to figure out the appropriate formulae yourself. For starters, it may be helpful to use the six steps of problem-solving and run through some example calculations by hand.

ImageFingerprint

The `ImageFingerprint` function creates a 64-bit image fingerprint using the specified hashing and resizing algorithm for a given colour image.

Inputs:

- An m-by-n-by-3 `uint8` array representing a colour image.
- A character vector containing either `'AvgHash'` or `'DiffHash'`.
- A character vector containing either `'Nearest'` or `'Box'`.

Output:

- A 1-by-64 `logical` row vector representing the image fingerprint.

Note: Assume $m \geq 8$ and $n \geq 9$ since the `ResizeBox` function can only downscale.

The `ImageFingerprint` function should call most of the previous functions to create a fingerprint. The order that it should call the functions is in Figure 1 (and highlighted throughout). However, we will repeat it:

1. Use the `GreyscaleLuma` function to turn the image into a greyscale image.
2. Resize the image with the `ResizeNearest` or `ResizeBox` functions:
   - If using the `AvgHash` function, resize the greyscale image to an 8-by-8 greyscale image.
   - If using the `DiffHash` function, resize the greyscale image to an 8-by-9 greyscale image.
3. Apply the `AvgHash` or `DiffHash` function.

Note: Mathematically, the ordering of the greyscaling and resizing does not matter. However, from a finite precision perspective, it does, so we will choose to use `GreyscaleLuma` first.

For example, we might use `ImageFingerprint` to create an image fingerprint as follows:

```
>> watch = imread("watch.png");
>> fp1 = ImageFingerprint(watch, 'DiffHash', 'Nearest');
>> DispFP(fp1);
10010011 00010110 00110110 10110110 01111000 00101011 11010000 10000110
```

Or:

```
>> watch = imread("watch.png");
>> fp2 = ImageFingerprint(watch, 'AvgHash', 'Box');
>> DispFP(fp2);
10001111 00000101 11100001 11100111 11100001 11000001 11000011 00011111
```

Tip: Stuck?

Remember, if you are having trouble with any of these functions, consider some of the following strategies:

- Try rereading the specification.
- Use the six steps of problem-solving.
- Try solving it by hand for a small example.
- Take a quick break.
- Try completing a different function.

FingerprintCollection

The `FingerprintCollection` function stores all of the image fingerprints from a list of image filenames so they can be quickly looked up for comparison.

Inputs:

- An m-by-1 `string` array representing a list of image file names.
- A character vector containing either `'AvgHash'` or `'DiffHash'`.
- A character vector containing either `'Nearest'` or `'Box'`.

Output:

- An m-by-1 `cell` array containing the fingerprint from each image file.

A fingerprint of an image will not change if we calculate it again (unless we change the fingerprinting algorithms we use). So it is more efficient to store each fingerprint than computing each fingerprint when we need to use it.

For the ith filename in the input `string` array, `FingerprintCollection` should read the image and call the `ImageFingerprint` function to obtain the image fingerprint and assign it to the ith index of the output `cell` array. Hence, every cell of the output `cell` array will be a 1-by-64 `logical` row vector.

Recall that we can assign and access `cell` array cells with the brace `{}` operators.

For example, we might use `FingerprintCollection` as follows to return a 4-by-1 `cell` array of our image fingerprints:

```
>> filenames = ["Duck1.jpg"; "Duck2.jpg"; "Rosie.jpg"; "RosieWithBall.jpg"];
>> fp_collection = FingerprintCollection(filenames, 'DiffHash', 'Box')

fp_collection =

  4×1 cell array

    {[1 0 1 1 0 0 1 1 1 1 1 0 1 0 1 0 1 1 1 0 0 1 1 1 0 1 1 0 0 ...  ]}
    {[1 0 0 0 0 1 0 1 1 1 0 1 0 0 1 0 1 1 0 0 0 1 1 1 1 1 1 0 0 ...  ]}
    {[0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0 1 0 1 1 1 0 0 1 0 1 1 1 1 ...  ]}
    {[1 0 1 1 1 1 1 0 1 1 1 0 0 0 0 0 1 1 0 1 1 0 0 0 1 1 0 1 0 ...  ]}
```

RankSimilarity

The `RankSimilarity` function displays n image files ranked by descending similarity (i.e., the image most similar to the search image will be at the top of the list).

Inputs:

- A 1-by-64 `logical` row vector representing the image fingerprint to search.
- An m-by-1 `string` array representing a list of image file names.
- An m-by-1 `cell` array containing a collection of image fingerprints.
- An integer n denoting how many image fingerprint matches to display.

Output:

- None

Note: Assume $1 \leq n \leq m$ and $1 \leq n < 100$.

The distance found using the `HammingDistance` function between our image and the image fingerprints in our fingerprint collection determines the similarity. The lower the Hamming distance between the two fingerprints, the more similar the two images are.

The `RankSimilarity` function should display the n images in the following form with rank *`r`* as an integer, hamming distance *`ham`* as an integer, and the filename *`filename`* as a string:

```
r. ham - filename
```

For the spaces between each of the above, there should be:

- Sufficient space before the rank *`r`* depending on n such that all the ranks align properly in a column (right-justified). For example, if $n = 8$, there should be no extra space. However, if $n = 88$, there should be a single space for the ranks 1 to 9 and zero for the rest.
- One or two spaces between the Hamming distance *`ham`* and the rank *`r`* (since the maximum Hamming distance is 64) so that it also aligns properly in columns (right-justified).
- One space between the hyphen and filename *`filename`*, (left-justified).

For example, if $n = 12$, we might obtain something (with a hypothetical dataset) as follows:

```
>>
   ⋮
 8.  4 - filename777
 9.  8 - filename2
10.  8 - filename5
11. 11 - filename34
12. 15 - file3
```

For example, we might use `RankSimilarity` as follows to find the top $n = 4$ similar images compared against the `DeckRosie.jpg` image:

```
>> filenames = ["Duck1.jpg"; "Duck2.jpg"; "Rosie.jpg"; "RosieWithBall.jpg"];
>> fp_collection = FingerprintCollection(filenames, 'DiffHash', 'Box')
>> deck_rosie = imread("DeckRosie.jpg");
>> deck_rosie_fp = ImageFingerprint(deck_rosie, 'DiffHash', 'Box');
>> RankSimilarity(deck_rosie_fp, filenames, fp_collection, 4);
1. 24 - Duck1.jpg
2. 27 - Duck2.jpg
3. 33 - RosieWithBall.jpg
4. 39 - Rosie.jpg
```

Note: Break ties by the indices (in ascending order) of the images in the fingerprint collection. For example, if our image has Hamming distance of 5 from the image in index `1` and `5`, we will rank the image in index `1` higher than image `5`.

Marking

The project is marked out 50.

All the functions are submitted to MATLAB Grader. There is a maximum of five submission attempts per function.

Functionality

Each of the 10 required functions will be marked out of 4 for functionality.

The weighting of passed tests will determine the functionality marks for each function submitted to MATLAB Grader. Some tests will be easier to pass than others. Even if you cannot complete all the functions, please submit the functions you have written to obtain partial marks.

The MATLAB Grader submission may not be set up immediately as we want to encourage you to test your own code rather than relying on MATLAB Grader to do it for you.

Style

One of the ten functions you have submitted will be selected randomly, and the best "functionality" solution will be marked out of 10 for style.

Style includes the following elements (each worth 2 marks):

- Descriptive header comment.
- Adequate body comments.
- Sensible variable names.
- Correct indentation (i.e., horizontal whitespace).
- Correct "code chunking" (i.e., vertical whitespace).

We expect all code submitted to MATLAB Grader to adhere to the above style elements. Remember that you only have five submission attempts for each function. It is unwise to leave adding good style until after you have submitted a fully working function, as writing stylish code from the outset will make it easier for you to debug and maintain your code.

The number of code lines for each function can differ wildly from one to 25 lines of code (not including comments). It is also perfectly fine if your project solution runs to several hundred lines of code as long as your code works and uses a good programming style.

Note: It is still possible to get marks for good style, even if your code does not work!

Questions?

If you have any questions regarding the project, please check through this document first. If it does not answer your question, feel free to ask it on the class forum.

Do **not** publicly post any of your project code (including MATLAB Grader test results). If your question requires that you include some of your code, make sure it is posted as a **private** thread.