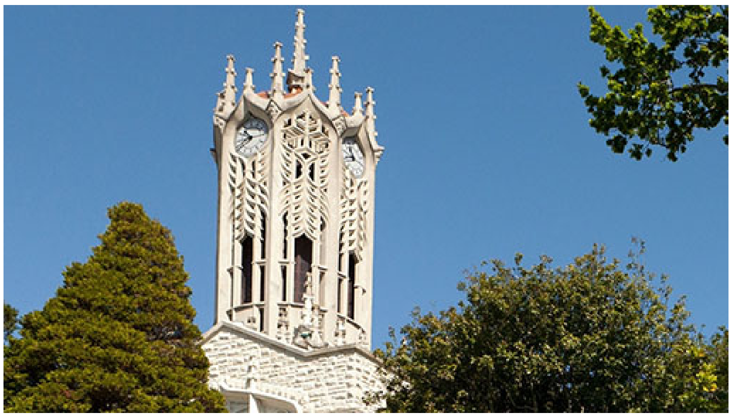
Creating k-colour images

An application of the k-means clustering algorithm

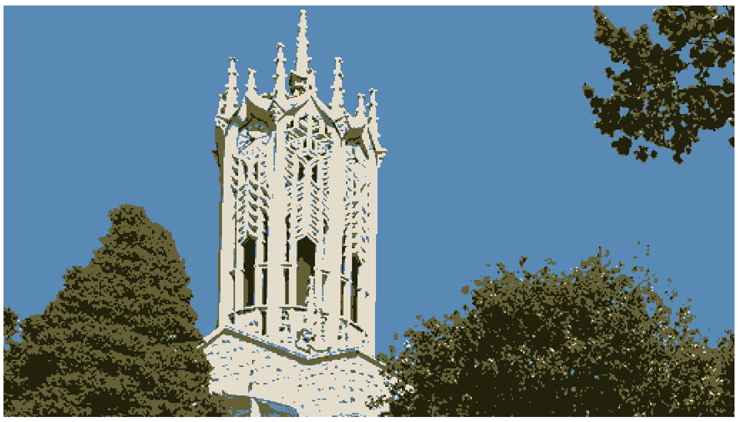
Project due date: You will need to upload your code before

11.59pm on Friday the 7th of September.



From hundreds, of colours…

To just a few (here k=4)



NOTE: This is version 2 of the project document. If you find a typo please post it to the Piazza project typo thread. If any further significant typos are found a newer version of the project will be uploaded to Canvas, with those typos corrected.

Introduction

This project requires you to write code that will allow you to create k-colour images (where k is a specified small number such as 4) from an image containing many hundreds (or even thousands) of colours. The idea is to use the k-means clustering algorithm to group the pixel colours used in the image into clusters. We can then determine the appropriate colours to use by taking the mean RGB values of each cluster, so that the reduced colour image is as accurate a representation of the original colour image as possible. On completion of the project you will have written code that can take any arbitrary image and produce a k-colour image where k is a value specified by the user (for practical reasons smaller images are recommended and k is likely to be in the range 2 to 16, although you are welcome to try larger images and values if you are happy to wait for your code to run).

It is expected that you will create your own small images and arrays to test your code (make sure you work out the answers on some small sets of test data by hand, so you know what you should get). Before the submission date you will be supplied with some test data and test scripts which can be used to help check your code. If your code fails some of the supplied test code there is obviously work to be done. If your code passes the supplied test scripts it is a good sign you are on the right track but please note that the supplied test code isn't identical to that used to mark your code. If you rely just on the supplied test code (without doing tests of your own) you will run the very real risk of submitting code that will not pass all the marking tests. Do NOT rely solely on the supplied test scripts, part of doing the project is to learn how to extensively test your own code on a range of different values.

This project is designed to take the average student around **15 hours** to code up but some people may find it takes upwards of 60 hours (and others will finish it in just a few hours). You will not know ahead of time how long it will take you, so please don't leave it to the last minute to make a start!

Remember the golden rule

Write your own code and don't give your code to anyone. Friends don't let friends share code. This project needs to be YOUR work, it is an individual project, not a group project. Copying and/or sharing code is academic misconduct. Please note that every project submission is passed through software that detects plagiarism so if you copy **you WILL be caught**. Unfortunately every year students discover the hard way that I'm not kidding. Let me repeat, DO NOT COPY: YOU WILL BE CAUGHT. If you give your code to another student and they copy it, you will both be guilty of misconduct. Copying or supplying code typically results in both the person who copied and the person who supplied the code being awarded a mark of zero for the project, as well as your names going on the academic misconduct register. If you have already been found guilty of academic misconduct on another course or assignment the penalty may be even greater (e.g. a fine or fail of the course).

To help you avoid academic misconduct I have put together a very short best practices guide which I recommend you read. It can be found in the project directory on Canvas.

What Matlab functions can I use?

You can use any functions we have covered in class and any other functions that are part of Matlab's **core** distribution (i.e. not functions from any toolboxes you may have installed). Note that for this project *the use of functions that are only available in toolboxes is not permitted*. 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 simply call some toolbox functions that do all the hard work for you (and in case you were wondering, the kmeans function in Matlab is part of a toolbox NOT part of the core Matlab distribution).

A list of some of the core Matlab functions can be found on pages 283 to 285 of the course manual. If you are uncertain whether a particular function is part of the Matlab's core distribution or not, type the following at the command line

```
which <function>
```

where the term *<function>* is replaced by the name of the function you are interested in. Check the text displayed 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 plot
built-in (C:\Matlab\R2017b\Pro\toolbox\matlab\graph2d\plot)
```

This shows us that the plot function is part of the standard core Matlab distribution (notice how the word matlab follows the word toolbox).

```
>> which kmeans
C:\Matlab\R2017b\Pro\toolbox\stats\stats\kmeans.m
```

This shows us that the kmeans function is part of the stats toolbox and is therefore not permitted to be used for the project (notice how the word stats follows the word toolbox).

How to tackle the project

Do not be daunted by the length of this document. It is long because a lot of explanation is given as to exactly what each function needs to do.

The best way to tackle a big programming task is to break it down into small manageable chunks. A lot of this work has already been done for you in the form of seven functions to write. Each function has a detailed description of what it needs to do. Have a read through the entire document and then pick a function to start on. Remember you don't have to start with `SelectKRandomPoints`, although this isn't as tricky as some of the other functions. You may prefer to start with one of the other straight forward functions such as `GetRGBValuesForPoints` or `SquaredDistance` (which can both be written in just a few lines of code).

Write as many of the functions as you can. When writing some of the specified functions you may find it useful to call one of the other functions you have written. You may even wish to write additional helper functions that have not been listed in the specifications. This is absolutely fine (in fact it is good coding practice to do so). Make sure you remember to submit any extra helper functions you write as they will be needed so that your code runs correctly. Please note that you will be able to submit up to 16 functions in total, which leaves room for up to 9 helper functions (which should be ample).

Several functions require careful thought, particularly those that form the heart of the k-means algorithm (remember those 5 steps of problem solving!). If you are having trouble understanding how a function should work, remember to work through the problem by hand with a small data set.

Note that I don't expect everyone to get through all of the functions. Some of them are relatively easy (e.g. `GetRGBValuesForPoints` and `SquaredDistance`) while others are quite tricky (e.g. `KMeansRGB`). You can still get a pretty good mark even if you don't complete all of the functions.

You will receive marks both for correctness (does your code work?), style (is it well written?) and execution time (how fast does it run?). (An "A" grade student should be able to nut out everything except perhaps full marks for execution time (it can be **extremely** challenging to get the algorithm running quickly on large images). B and C grade students might not get a fully working solution. Even if you only get three of the functions working, you can still get over 50% for the project, as long as the code you submit is well written (since you get marks for using good style).

An overview of the k-means algorithm.

Before we delve into some more detail, let's take a quick high level outline of how the k-means algorithm works.

The k-means clustering algorithm is an iterative refinement technique that allows for a set of data values to be partitioned into k clusters (where k is typically a small number such as 4).

Given an initial set of k means (which can be chosen at random) the algorithm proceeds by alternating between two steps

- **Assignment step**: every data point is assigned to a cluster based on which of the means it is closest to
- **Update step**: new means are calculated for each cluster (i.e. the mean is calculated from all the points currently determined to be in that cluster)

The above steps are repeated until the means found in the update step are the same as the previous set of means, at which point the algorithm has converged. Note that convergence is NOT guaranteed, although in practice it isn't common to get stuck in an infinite loop.

For more on k-means see: https://en.wikipedia.org/wiki/K-means_clustering

Here is an example of a k-means algorithm applied to a very small 6 element 1D data set, with k=3.

Our data set is the list of values [40 50 90 100 200 210]

We pick three values at random to be our initial seed means, e.g. the 1st, 2nd and 4th value.

Our seed means are therefore [40, 50, 100]

Now we proceed to the **assignment step**, assigning each point to either cluster 1, cluster 2 or cluster 3, depending on whether the point is closest to mean 1, mean 2 or mean 3 (in practice we calculate the squared distance between each point and all the means to determine the closest mean – using the squared distance saves us having to take the square root of the distance, which speeds things up a little).

40 is closest to mean 1. 50 is closest to mean 2. The remaining data points are all closest to mean 3.

Data:	[40	50	90	100	200	210]
Cluster:	[1	2	3	3	3	3]

We can now **update the means** by calculating new means for each of the three clusters.

The mean of cluster 1 is 40 (as that is the only point in cluster 1). The mean of cluster 2 is 50 as that is the only point in cluster 2). The mean of cluster 3 is (90+100+200+210)/4=150

Our new means are therefore [40, 50, 150]

Now we proceed again to the **assignment step**, reassigning each point to either cluster 1, cluster 2 or cluster 3, depending on whether the point is closest to mean 1, mean 2 or mean 3.

40 is closest to mean 1. 50 and 90 are closest to mean 2. 100 is tied (as it is 50 units from mean 2 and also 50 units from mean 3). In the event of a tie we will assign it to the lowest cluster number (in this instance 2). 200 and 210 are closest to mean 3.

Data:	[40 50 90 100 200 210]
Cluster:	[1 2 2 2 3 3]

We can now **update the means** by calculating new means for each of the three clusters.

The mean of cluster 1 is still 40 (as that is the only point in cluster 1). The mean of cluster 2 is now (50+90+100)/3 = 80. The mean of cluster 3 is (200+210)/2=205

Our new means are therefore [40, 80, 205]

Now we proceed again to the **assignment step**, reassigning each point to either cluster 1, cluster 2 or cluster 3, depending on whether the point is closest to mean 1, mean 2 or mean 3

40 and 50 are closest to mean 1. 90 and 100 are closest to mean 2. 200 and 210 are closest to mean 3.

Data:	[40 50 90 100 200 210]
Clusters:	[1 1 2 2 3 3]

We can now **update the means** by calculating new means for each of the three clusters.

The mean of cluster 1 is (40+50)/2=45. The mean of cluster 2 is now (90+100)/2 = 95. The mean of cluster 3 is (200+210)/2=205

Our new means are therefore [45, 95, 205]

Now we proceed again to the **assignment step**, reassigning each point to either cluster 1, cluster 2 or cluster 3, depending on whether the point is closest to mean 1, mean 2 or mean 3.

40 and 50 are closest to mean 1. 90 and 100 are closest to mean 2. 200 and 210 are closest to mean 3. Note that none of the cluster assignments have changed this time around.

We proceed as usual to **update the means** by calculating new means for each of the three clusters.

The mean of cluster 1 is (40+50)/2=45. The mean of cluster 2 is now (90+100)/2 = 95. The mean of cluster 3 is (200+210)/2=205

Our new means are therefore [45, 95, 205] and NOTE THIS IS UNCHANGED from last time.

Once our means no longer change, the algorithm has converged and data points will no longer be moved between clusters. We can therefore stop our calculations.

Note that they above description describes how you the algorithm works with 1D data, you will be applying the algorithm to 3D data. The principle is the same but determining which mean a point is closest to is a little trickier as your data points will be points in 3D space, as will your means.

Overview of Functions to Write

There are seven in total:

- `SelectKRandomPoints` (fairly easy)
- `GetRGBValuesForPoints` (fairly easy)
- `SquaredDistance` (easy, useful when writing `AssignToClusters`)

- `AssignToClusters` (fairly challenging, will be used by `KMeansRGB`)
- `UpdateMeans` (fairly challenging, will be used by `KMeansRGB`)
- `KMeansRGB` (fairly challenging, the heart of the project)
- `CreateKColourImage` (easier)

`SelectKRandomPoints`, `GetRGBValuesForPoints`, `KMeansRGB` and `CreateKColourImage` are all called directly from the supplied main script file. You will need to implement these functions before being able to use the supplied script file to generate k-colour images.

The remaining three functions mentioned above are not called directly from the supplied main script file but should prove useful to call when writing some of the other functions (even if you decide not to call them in this way you should still write them as they will be marked).

`SquaredDistance` will likely be handy to call from your `AssignToClusters` function. `AssignToClusters` and `UpdateMeans` will be useful when writing your `KMeansRGB` function.

The pages following have detailed specifications of what each function should do. Note that while the functions are designed to work together with the supplied main script to allow the creation of a k-colour image, many of the functions can be written and tested in isolation. Even if you get stuck on one function you should still try and write the others.

SelectKRandomPoints

Purpose	`SelectKRandomPoints` generates a list of k randomly selected pixels from an image.
Input(s)	It takes **two** inputs in the following order: 1) A 3D image array from which to select points from 2) The number of points to randomly select
Output(s)	It returns a **single** output, a 2D array, containing k rows and 2 columns, representing k randomly selected points (pixels).

Every row in the output array identifies the location of a point (i.e. pixel) in the image. Each location is specified in terms of a row and column index, hence each row contains 2 values. E.g. if 3 points were requested, the output would contain 3 rows and 2 columns.

The first column of the output array contains the row indices of the randomly selected points and the second column of the output array contains the column indices of the randomly selected points. Each row of the output array should correspond to a unique randomly selected point (i.e. no points are repeated). For example, if the output was the following 3 row, 2 column 2D array:

```
244  213
127   40
127  117
```

The first point is in row 244, column 213 of the image.
The second point is in row 127, column 40 of the image.
The third point is in row 127, column 117 of the image. Note that while the row index for this point is the same as the second point, it is a different point as the column index is different.

Keep the following in mind:

- Note the capital letters in the name. Case matters in Matlab and you should take care that your function names EXACTLY match those in the projection specification.
- It should be possible for any point within the image to be randomly selected with equal likelihood.
- The list of points returned should include no duplicates, e.g. if 10 points are requested, all 10 points should be unique (you should not see any rows in the output that contain identical values)
- The list of points should not include points outside the range of the image. E.g. if the image contains 256 rows and 256 columns of pixels you would not expect to generate a point with row index 257, as this would be outside the range of the image.

Example calls

Here are some examples of calls to `SelectKRandomPoints` assuming that A is a 256x256x3 array representing an RGB image,

```
>> points = SelectKRandomPoints(A,4)

points =

   256   113
   129    40
   129   147
   188   157

>> points = SelectKRandomPoints(A,6)

points =

   150   192
   219    40
     9    37
   225   154
   103    65
    10    40
```

Here are some examples of calls to `SelectKRandomPoints` assuming that B is a 2x2x3 array representing an RGB image,

```
>> points = SelectKRandomPoints(B,1)

points =

     2      2


>> points = SelectKRandomPoints(B,4)

points =

   1    2
   2    2
   2    1
   1    1
```

Note that with the last function call, as there are only 4 points to choose from, we expect to get all 4 points (as no duplicates can be returned). You may assume that your `SelectKRandomPoints` will never be called with a request for more points than are available.

The GetRGBValuesForPoints function

Purpose	Returns the RGB colour values for a list of specified points from an image
Input(s)	It takes **two** inputs in the following order: 1) A 3D image array from which to fetch RGB values from 2) A 2D array of k rows and 2 columns identifying which points (i.e. pixels) to extract colour values for.
Output(s)	It returns a **single** output, a 3D array, containing k rows, 1 column and 3 layers, representing a list of k points from the image (each row corresponds to the colour information for a particular pixel from the image).

Every row in the second input array identifies the location of a point (i.e. a pixel) in the image. Each location is specified in terms of a row and column index, hence each row contains 2 values.

The first column contains the row indices of the points to extract colour information for.
The second column contains the column indices of the points to extract colour information for.

e.g. if the 2D array of points contained the following values

```
1  1
2  3
4  3
```

We would be extracting the colour values for the pixel located in row 1, column 1 of the image, the pixel located in row 2, column 3 of the image and the pixel located in row 4, column 3 of the image.

Worked example

Suppose that the image contained the following values

```
A(:,:,1) =

   192   227    38   208
    65   245    66    62
   129   140   214   237
   178    35    65    89
```

```
A(:,:,2) =

    50    90   234    97
    64   212    73   145
   157   149   193    19
   121   140   192    14
```

```
A(:,:,3) =

   135   145    41    42
   199   120   203   154
   238     3    79    67
    33    86   135   167
```

If the pixels we are interested in are specified as

```
P =

     1     1

     2     3

     4     3
```

Then we would extract the RGB values for the pixel in row 1, column 1 (i.e. 192, 50 and 135)

The RGB values for the pixel in row 2 column 3 (i.e. 66, 73, 203)

And the RGB values for the pixel in row 4 column 3 (i.e. 65, 192, 135)

We would create a 3D array of Colours to return containing the following values:

```
Colours(:,:,1) =

   192
    66
    65
```

```
Colours(:,:,2) =

    50
    73
   192
```

```
Colours(:,:,3) =

   135
   203
   135
```

Example calls

Here are some examples of calls to `GetRGBValuesForPoints`

```
>> A = imread('VanuatuSmall.jpg') % read in an image
>> P = [1 1] % row and column values for the single point to get
>> % fetch the colour data for the pixel in row 1, column 1
>> Colours = GetRGBValuesForPoints(A,P)


>> B = round(rand(4,4,3)*255) % generate small random 4x4 pixel RGB image
>> P = [1 1;2 3; 4 3] % row and column values for the 3 points to get
>> Colours = GetRGBValuesForPoints(B,P) % fetch the colour data
```

The SquaredDistance function

Purpose	Calculates the square of the distance between two points in 3D space
Input(s)	It takes **two** inputs in the following order: 1) An array containing three elements representing a point in 3D space 2) An array containing three elements representing a second point in 3D space
Output(s)	A single output, the square of the distance between the two points in 3D space.

Notes

The distance referred to is the geometric straight line distance between the two points when interpreting them as points in 3D space. This function returns the SQUARE of this distance, so mathematically it is equivalent to the following calculation for two 3D points, P and Q

$$D = (P_1 - Q_1)^2 \ + \ (P_2 - Q_2)^2 + (P_3 - Q_3)^2$$

Note that the colour values for a pixel can be interpreted as a point in 3D space $P = (P_1, P_2, P_3)$

The three coordinates are the colour values for the pixel, i.e. the first coordinate P_1 is the amount of red, the second P_2 is the amount of green and the third P_3 is the amount of blue.

You may assume that both P and Q will both be the same size.

Your code should be written to handle the case where the two points P and Q are 1D arrays (e.g. both having 1 row and 3 columns or both having 3 rows and 1 column)

In addition your code should also handle the case where P and Q are both 1x1x3 arrays (i.e. a point may be passed in as a 3D RGB image having 1 row 1 column and 3 layers, where each layer contains a colour value).

Worked example

Suppose we had the two points

$P = (0,0,0)$ and $Q = (3,4,0)$

Then the squared distance would be given by

$$D = (0-3)^2 \ + \ (0-4)^2 + (0-0)^2$$

$$D = 9 + 16 = 25$$

Note that the square distance can never be negative (as it is a squared value).

Example calls

Here are some examples of calls to `SquaredDistance`

```
>> P = [0 0 0] % this as a 1x3 1D array
>> Q = [255 255 255] % this as a 1x3 1D array

>> % find the squared distance between P and itself
>> squaredDistance = SquaredDistance(P,P)
```

This will produce the result:

```
squaredDistance =

      0

>> % find the squared distance between P and Q
>> squaredDistance = SquaredDistance(P,Q)
```

This will produce the result:

```
squaredDistance =

      195075
```

```
>> P = [0; 0; 0] % this as a 3x1 1D array

>> Q = [3; 4; 0] % this as a 3x1 1D array

>> % find the squared distance between P and Q

>> squaredDistance = SquaredDistance(P,Q)
```

This will produce the result:

```
squaredDistance =

      25
```

Suppose we have a 3D array of colours containing the following values:

Colours(:,:,1) =	Colours(:,:,2) =	Colours(:,:,3) =
192	50	135
66	73	203
65	192	135

Then we could calculate the squared distance between the first pixel and the third pixel as follows:

```
>> P = Colours(1,1,:) % this is the first pixel, in row 1 column 1
>> Q = Colours(3,1,:)% this is the third pixel in row 3, column 1

>> % find the squared distance
>> squaredDistance = SquaredDistance(P,Q)
```

This will produce the result:

```
squaredDistance =

        36293
```

The AssignToClusters function

Purpose	Assigns each point (pixel) in an image to a cluster, based on which mean that point is closest to.
Input(s)	It takes **two** inputs in the following order: 1) A 3D array with m rows, n columns and 3 layers, containing an RGB image 2) A 3D array containing k rows, 1 column and 3 layers containing the colour information for each of k means.
Output(s)	A single output being a 2D array with m rows and n columns, that contains the corresponding cluster number for each pixel in the image. i.e. the value in row 1 column 1 of the output will identify the cluster for the pixel in row 1 column 1 of the image. This value is determined by identifying which of the k means this pixel was closest to.

Notes

To find which cluster a pixel belongs to you will need to compute the squared distance between the pixel and each of the k means (which are stored in different rows of the second input) and then choose the mean that is closest (i.e. the one that has the least squared distance). In the event of a tie the lowest cluster number will be chosen.

If for example there are three means to choose from and a pixel was closest to the third mean, it would be assigned to cluster 3. Similarly a pixel that was closest to the second mean would be assigned to cluster 2. The means are numbered according to the row number they are stored in (so mean 3 is stored in the third row of the second input argument).

Worked example

Suppose that we have an image containing the following values

```
A(:,:,1) =

   192   227    38   208
    65   245    66    62
   129   140   214   237
   178    35    65    89
```

```
A(:,:,2) =

    50    90   234    97
    64   212    73   145
   157   149   193    19
   121   140   192    14
```

```
A(:,:,3) =

   135   145    41    42
   199   120   203   154
   238     3    79    67
    33    86   135   167
```

And we have a 3D array of means containing the following values:

```
Means(:,:,1) =

   192
    66
    65
```

```
Means(:,:,2) =

    50
    73
   192
```

```
Means(:,:,3) =

   135
   203
   135
```

The first pixel in row 1, column 1 has the RGB colour values (192,50,135)

We have three means, with colour values as follows:

Mean 1 (from row 1) has colour values (192,50,135)

Mean 2 (from row 2) has colour values (66,73,203)

Mean 3 (from row 3) has colour values (65,192,135)

Calculating squared distances between the pixel in row 1 column 1 and each of these gives the following squared distances

- Squared distance 1 is 0
- Squared distance 2 is 21029
- Squared distance 3 is 36293

Clearly the smallest squared distance is 0, so the pixel in row 1, column 1 belongs to cluster 1.

We therefore assign the value of the output array in row 1, column 1 to be equal to 1.

Now consider the pixel in row 2, column 1 which has the colour value (65,64,199)

Calculating squared distances between the pixel in row 1 column 1 and each of these gives the following squared distances

- Squared distance 1 is 20421
- Squared distance 2 is 98
- Squared distance 3 is 20480.

Clearly the smallest squared distance is 98, so the pixel in row 2, column 1 belongs to cluster 2.

We therefore assign the value of the output array in row 2, column 1 to be equal to 2.

Example calls

Here is an examples of a call to `AssignToClusters`, assuming the same A and Means array as used in the worked example.

```
>>[clusters] = AssignToClusters(A,Means)

clusters =

     1     1     3     1
     2     1     2     3
     2     3     1     1
     1     3     3     2
```

The UpdateMeans function

Purpose	Calculate the mean values for each cluster
Input(s)	It takes **three** inputs in the following order: 1) A 3D array with m row, n columns and 3 layers, containing an RGB image 2) A single value, k, specifying how many clusters there are. 3) A 2D array with m rows and n columns specifying which cluster each pixel belongs to.
Output(s)	A single output, a 3D array containing k rows, 1 column and 3 layers, containing the mean values for each cluster. E.g. the mean R, G and B values for cluster 1 will be stored in row 1. The mean R, G and B values for cluster 2 will be stored in row 2 and so on.

Worked example

Suppose that we have an image containing the following values

A(:,:,1) =	A(:,:,2) =	A(:,:,3) =

192 227 38 208
65 245 66 62
129 140 214 237
178 35 65 89
	50 90 234 97
	64 212 73 145
	157 149 193 19
	121 140 192 14
		135 145 41 42
		199 120 203 154
		238 3 79 67
		33 86 135 167

The data has been divided into 3 clusters and each pixel has been allocated to a cluster as follows:

```
clusters =

     1     1     3     1

     2     1     2     3

     2     3     1     1

     1     3     3     2
```

To determine the mean R, G and B values for cluster 1 we need to average the Red colour values for all pixels from cluster 1 (these values are shown in red above).

Our mean red value for cluster 1 is therefore (192+227+208+245+214+237+178)/7 = 214.4286

This value will be stored in row 1, column 1, layer 1 of the output array.

Our mean green value for cluster 1 is therefore (50+90+97+212+193+19+121)/7 = 111.7143

This value will be stored in row 1, column 1, layer 2 of the output array.

Our mean blue value for cluster 1 is therefore (135+145+42+120+79+67+33)/7 = 88.7143

This value will be stored in row 1, column 1, layer 3 of the output array.

Example calls

Here is an example of calling UpdateMeans using the array A and the array clusters from the worked example.

```
>> means = UpdateMeans(A,3,clusters)

means(:,:,1) =

  214.4286
   87.2500
   68.0000


means(:,:,2) =

  111.7143
   77.0000
  172.0000


means(:,:,3) =

   88.7143
  201.7500
   83.8000
```

The KMeansRGB function

Purpose	Partition the points in an image into k clusters, using the k-means algorithm to do so.
Input(s)	It takes **three** inputs in the following order: 1) A 3D array with m rows, n columns and 3 layers, containing an RGB image. 2) A 3D array containing k rows, 1 column and 3 layers, containing the seed mean values which will be used to initialise the k-means algorithm. 3) The maximum number of iterations to perform.
Output(s)	It returns **two** outputs in the following order: 1) A 2D array with m rows and n columns specifying which cluster each pixel belongs to 2) A 3D array containing k rows, 1 column and 3 layers, where each row contains the mean colour values for the cluster corresponding to that row number. E.g. the mean R, G and B values for cluster 1 will be stored in row 1. The mean R, G and B values for cluster 2 will be stored in row 2 and so on.

Notes

Recall that the k-means algorithm works as follows:

Given an initial set of k means (sometimes called the seed means) the algorithm proceeds by alternating between two steps

- **Assignment step**: every data point is assigned to a cluster based on which of the means it is closest to. Hint: you should use your `AssignToClusters` function to do this.
- **Update step**: new means are calculated for each cluster (i.e. the mean is calculated from all the points currently determined to be in that cluster). Hint: you should use your `UpdateMeans` function to do this.

The above steps are repeated until the means found in the update step are the same as the previous set of means, at which point the algorithm has converged and the cluster and mean information returned.

Note that convergence is NOT guaranteed, although in practice it isn't common to get stuck in an infinite loop. To prevent an infinite loop you should stop looping after the maximum number of iterations specified. If the maximum number of iterations is reached without convergence being achieved, a warning message should be displayed to the user stating "Maximum number of iterations reached before convergence was achieved". In this instance your code should then return the cluster and mean information that was found after the maximum number of iterations.

This function is the heart of the project and you may find it fairly challenging. It is also the function that the least amount of help is given with. Before attempting it, it is recommended you first write `AssignToClusters` and `UpdateMeans`, as you will want to call those functions from your KMeansRGB function.

If you have already written `AssignToClusters` and `UpdateMeans` the amount of code required for KMeansRGB can easily fit on a page though.

Example calls

Here is an example of a call to KMeansRGB

```
>> [ clusters, means ] = KMeansRGB(A,seedmeans,100)
```

The CreateKColourImage function

Purpose	To create a k-colour image for an image that has had its pixels divided into k clusters. All pixels in a given cluster will be recoloured using the mean colour values for that cluster.
Input(s)	It takes **two** inputs in the following order: 1) A 2D array with m rows and n columns, specifying which cluster each pixel belongs to 2) A 3D array containing k rows, 1 column and 3 layers, where each row contains the mean colour values for the cluster corresponding to that row number. E.g. the mean R, G and B values for cluster 1 will be stored in row 1. The mean R, G and B values for cluster 2 will be stored in row 2 and so on.
Output(s)	It returns a single output, A 3D array of unsigned 8 bit integers with m rows, n columns and 3 layers, representing an RGB image. The colour of each pixel in the image is determined by the colour associated with that cluster. E.g. all pixels in cluster 1 will be coloured using the RGB values from row 1 of the second input (i.e. the mean colour values associated with cluster 1).

Notes

It is important to ensure you return an array of unsigned 8 bit integers. You should ensure the array returned is of type **uint8** and remember that as such it will contain only integer values between 0 and 255 inclusive.

The second input passed into `CreateKColourImage` will contain mean colour values and it is very likely that some of these will not be integers. All non-integer values should be **rounded to the nearest integer**.

Worked example

Suppose we had the following 2D cluster array

```
clusters =

     1     2

     2     1
```

with the following 3D array of cluster means

`means(:,:,1) =`
`     0`
`   255`
	`means(:,:,2) =`
	`     0`
	`   255`
		`means(:,:,3) =`
		`     0`
		`   255`

Here row 1 corresponds to the mean values for cluster 1 (which in this case is the colour black)

Row 2 corresponds to the mean values for cluster 2 (which in this case is the colour white)

The resulting image would have black pixels in the top left and bottom right corner and white pixels in the bottom left and top right, as follows:

```
MyImage(:,:,1) =            MyImage(:,:,2) =            MyImage(:,:,3) =

     0   255                     0   255                     0   255
   255     0                   255     0                   255     0
```

Example calls

Here is an example of a calls to `CreateKColourImage`, assuming the cluster data and means data as given in the worked example:

```
>> MyImage = CreateKColourImage(clusters,means)
```

This would return:

```
MyImage(:,:,1) =

     0   255
   255     0


MyImage(:,:,2) =

     0   255
   255     0


MyImage(:,:,3) =

     0   255
   255     0
```

How the project is marked

A mark schedule and some test scripts will be published prior to the due date, outlining exactly how marks will be allocated for each part of the project and giving you the opportunity to test your code. You will receive marks both for correctness (does your code work?), style (is it well written?) and execution time (how fast does it run?).

Each of the **seven** required function will be marked independently for correctness, so even if you can't get everything to work, please do submit the functions you have written. Some functions may be harder to write than others, so if you are having difficulty writing a function you might like to try working on a different function for a while and then come back to the harder one later.

Note it is still possible to get marks for good style, even if your code does not work at all! Style includes elements such as using sensible variable names, having header comments, commenting the rest of your code, avoiding code repetition and using correct indentation.

Each function can be written in less than 20 lines of code (not including comments) and some functions can be written using just a few lines but do not stress if your code is longer. It is perfectly fine if your project solution runs to several hundred lines of code, as long as your code works and uses good programming style.

How the project is submitted

Submission is done by uploading your code to the Aropa website. More information will be provided on how to submit the project to Aropa in a Canvas email announcement. If you want to be out of Auckland over the break and still want to work on the project that will be perfectly fine, as long as you have access to a computer with Matlab installed and can access the internet to upload your final submission.

Submission Checklist

Here is a list of the **seven** functions specified in this document. Remember to include all seven (or as many of them as you managed to write) in your project submission. The filenames should be EXACTLY the same as shown in this list (including case). They should also work EXACTLY as described in this document (**pay close attention to the prescribed inputs and outputs and their order**). In addition if your functions call any other "helper" functions that you may have written, remember to include them too (you can submit up to a total of 16 files, so may include up to 9 helper functions). The seven required functions are:

- `SelectKRandomPoints`
- `GetRGBValuesForPoints`
- `SquaredDistance`
- `AssignToClusters`
- `UpdateMeans`
- `KMeansRGB`
- `CreateKColourImage`

Any questions?

If you have any questions regarding the project please first check through this document. If it does not answer your question then feel free to ask the question on the class Piazza forum. Remember that you should **NOT** be publicly posting any of your project code on Piazza, so if your question requires that you include some of your code, make sure that it is posted as a **PRIVATE** piazza query.

Have fun!