Project 1: Colour Compression

Tong Gia Huy [1★]
[1] *Faculty of Information Technology, VNUHCM-University of Science, Vietnam*
[★] *Student ID: 21127307, E-mail: tghuy212@clc.fitus.edu.vn*

Abstract

A digital image, in its core form, is a matrix of pixels. Each pixel, depending on different types digital image can have various structures. For a binary image, each pixel can either be 0 or 1, which corresponds to the colour black and white. A common colour scheme, *RGB* takes 3 values per pixel each ranging from 0 to 255, to represent the intensity of the Red, Green, and Blue making up the colour of that pixel. Because of this, the cost for storing a coloured image can be quite high. Therefore the problem calls for a way to reduce the number of colours in the picture while retaining as much detail as possible. By using k-means clustering, a powerful algorithm used to group data points into k number of clusters, it is possible to compress the colours in a picture down to no more than k values. In this project, I will implement k-means clustering to reduce the number of colours of an input image. This document will present the key ideas of the algorithm, as well as explain the instruction and necessary concepts to successfully implement it.

1 INTRODUCTION

K-means algorithm is used extensively in machine learning and data science. The term k-means was first used by James MacQueen in 1967, which he described it as a process for partitioning an N-dimensional population into k sets on the basis of a sample. In order to implement k-means in Python, it is necessary to first go through the details of this algorithm. In Section 2, I will give go through the idea and steps of the algorithm as well as explain some key concepts for calculation.

For the sample image, I chose the *Pillars of Creation* photo taken by the Hubble Space Telescope in 2014. Figure 1 illustrates the original and reduced image with $k = 4$ side-by-side.

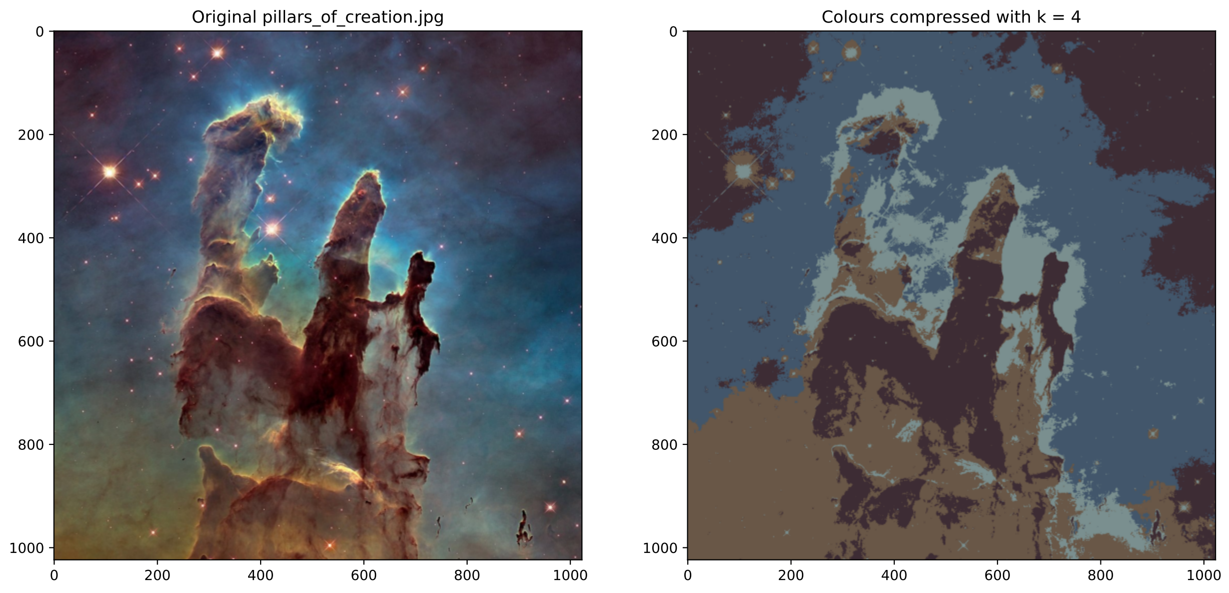
Original pillars_of_creation.jpg
Colours compressed with k = 4
0
200
400
600
800
1000
0
200
400
600
800
1000

Figure 1. The *Pillars of Creation*

2 ALGORITHM

2.1 The idea of k-means and its application in compressing colours

In this section, I will briefly go through the method of calculation used in this project. Step 1 and 2 are the general k-means clustering algorithm, Step 3 is an extra procedure used to produce the final compressed image. Additionally, in this context, the term *data point* and *pixel* can be used interchangeably.

- **Step 1**: Initialize a set of k clusters by generating k centroids. In this project, centroids are either randomized or chosen from the input data points.
- **Step 2**: In a loop, do the following:
  - Calculate the Euclidean distance between each data point and the clusters' centroids.
  - Assign the data points to their closest centroid.
  - Calculate the new centroids by obtaining the mean of the data points inside that cluster.
  - If the new centroids and current centroids converge, break out of the loop. If not then update the centroids and repeat until they converge.
- **Step 3**: For each pixel of the original image, replace its value with its cluster's centroid value.

I believe it is also worth mentioning that k-means does not always return exactly k valid clusters as it is still possible for a cluster to be empty, meaning there are no data point assigned to that cluster.

2.2 Key concept for NumPy calculations

The *NumPy* package (Harris et al. 2020) provides a powerful method of computation for matrices and higher-dimensional arrays. In this project, *NumPy* is used extensively during k-means clustering, especially in Euclidean distance calculation as the input data can be quite overwhelming for Python's native loop. Knowing how to make use of *NumPy*'s feature can improve the performance of the code drastically (riturajsaha 2021).

One such neat feature of *NumPy* is array broadcasting, which is a term used to describe how the package treat different shape arrays during operations (NumPy 2023). In order to understand how this works, let us look at how it is used to calculate the distances between centroids and pixels in an *RGB* image:

```
# Calculate the Euclidean distances between each data point and centroids.
distances = np.sqrt(np.sum(np.square(img_1d - centroids[:, np.newaxis]), axis=2))
```

where *img_1d* and *centroids* are of *numpy.ndarray* type. Each element inside the array is a pixel composed of 3 values for the intensity of Red, Green, and Blue.

For easier notation, let us define each pixel as:

$$RGB = \begin{bmatrix} R & G & B \end{bmatrix} \tag{1}$$

where R, G, and B are the intensity of each colour. Thus, our *img_1d* and *centroids* arrays would be defined as:

$$img\_1d = \begin{bmatrix} RGB_1 & RGB_2 & RGB_3 & \dots & RGB_N \end{bmatrix} \tag{2}$$

$$centroids = \begin{bmatrix} RGB'_1 & RGB'_2 & RGB'_3 & \dots & RGB'_K \end{bmatrix} \tag{3}$$

where N is the number of pixels and K is the number of clusters. It is important to note that the shape of *img_1d* and *centroids* are $(N, 3)$ and $(K, 3)$ respectively since we defined each pixel as an array in Eq. 1.

Now, because $N \neq K$ and most of the time neither are equal to 1, it is not possible for *NumPy* to broadcast the 2 arrays, making calculations impossible. This is where the *[:, np.newaxis]* command come into place. What it essentially does in this context is increase the dimension of the array (Huang 2015). This new dimension is added on the second axis, which is like creating new arrays encapsulating each of the original *RGB* pixel. With this operation, the *centroids*' pixels can be broadcasted along the new axis (Eq. 5) since its shape is now $(K, 1, 3)$.

Let us visualize how the arrays will behave during later operations, *img_1d* will expand downwards K times:

$$img\_1d' = \left[\begin{array}{ccccc} RGB_{11} & RGB_{12} & RGB_{13} & \dots & RGB_{1N} \\ \hline RGB_{21} & RGB_{22} & RGB_{23} & \dots & RGB_{2N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ RGB_{K1} & RGB_{K2} & RGB_{K3} & \dots & RGB_{KN} \end{array}\right] \tag{4}$$

and *centroids* will be stretched sideways N times:

$$centroids' = \left[\begin{array}{c|cccc} RGB'_{11} & RGB'_{12} & RGB'_{13} & \dots & RGB'_{1N} \\ RGB'_{21} & RGB'_{22} & RGB'_{23} & \dots & RGB'_{2N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ RGB'_{K1} & RGB'_{K2} & RGB'_{K3} & \dots & RGB'_{KN} \end{array}\right] \tag{5}$$

Notice that in Eq. 4 and 5, the shape of the 2 arrays are now matched, which allows calculation to be performed element-by-element. The rest of that line of code computes the Euclidean distance following the formular in Eq. 6. The result is an array where each row corresponds to a *centroid*, each column corresponds to a *pixel*, and each element is the distance between a pixel and a centroid.

$$d_{ij} = \sqrt{(R_{ij} - R'_{ij})^2 + (G_{ij} - G'_{ij})^2 + (B_{ij} - B'_{ij})^2}, \forall i \leq K, \forall j \leq N \tag{6}$$

The logic explained above also stays true for other colour model. There are different methods on how to implement broadcasting and distance calculation, but at the moment, I try to focus on one approach to better understand the basis of the package's calculation process.

2.3 Python implementation and explanation

My detailed implementation in *Python* is presented in the included *Notebook* file, this section will cover the purpose of each function inside the program. The prototype for each function is demonstrated bellow:

```python
def generate_centroids(data, n_chan, k, mode="random"):
    pass
```

```python
def k_means_clustering(img_1d, clusters, max_iter=1000, init_centroids="random",
    quiet=False):
    pass

def compress_colours(pixels, image_size, K_clusters, init_mode="random"):
    pass

def handle_input():
    pass

def main():
    pass
```

- **`generate_centroids(...)`**: Generate k initial centroids. Each centroid is either randomized throught mode `random` or picked from the original data points via mode `in_pixels`. This function implements Step 1 of Section 2.1.

- **`k_means_clustering(...)`**: The main function for calculation. It groups colour pixels into k clusters via k-means algorithm. This function implements Step 2 of Section 2.1. The `max_iter` parameter defines the upper iteration limit, if the number of computational loop exceed this value, the function will automatically return the current centroids and labels, regardless of whether true convergence occurred or not..

- **`compress_colours(...)`**: This function act like a bridge between user input and the calculation layer. It will take in the original image, pass it to the k-means algorithm, then use the centroids and label to reduce the number of colour of that image as described in Step 3 of Section 2.1. The function will then return the compressed image.

- **`handle_input()`**: Handle user input. The function is responsible for loading the image, as well as receiving the number of clusters, centroid initialization mode and output file type from the user.

- **`main()`**: Execute the entire program. It will first handle the input, compress the image's colours with k-means, then display the original and compressed image side by side with *MatPlotLib* (Hunter 2007), as well as save the compressed image to a user-defined file type.

3 RESULTS AND DISCUSSION

I ran k-means on the *Pillars of Creation* image with 3 values of k: 3, 5 and 7. For each k, the clustering algorithm used both *random* and *in_pixels* initialization scheme. The best result where the program return exactly k colours is shown in Figure 2.

As mentioned in Section 2.1, due to the nature of the algorithm and the selection of the first centroids, we do not always receive the expected k colours as some cluster have a possibility of becoming empty and thus discarded by the program. This happen when one or more centroids are positioned too far away from the data points, because of this, those centroids are not assigned to any data point.

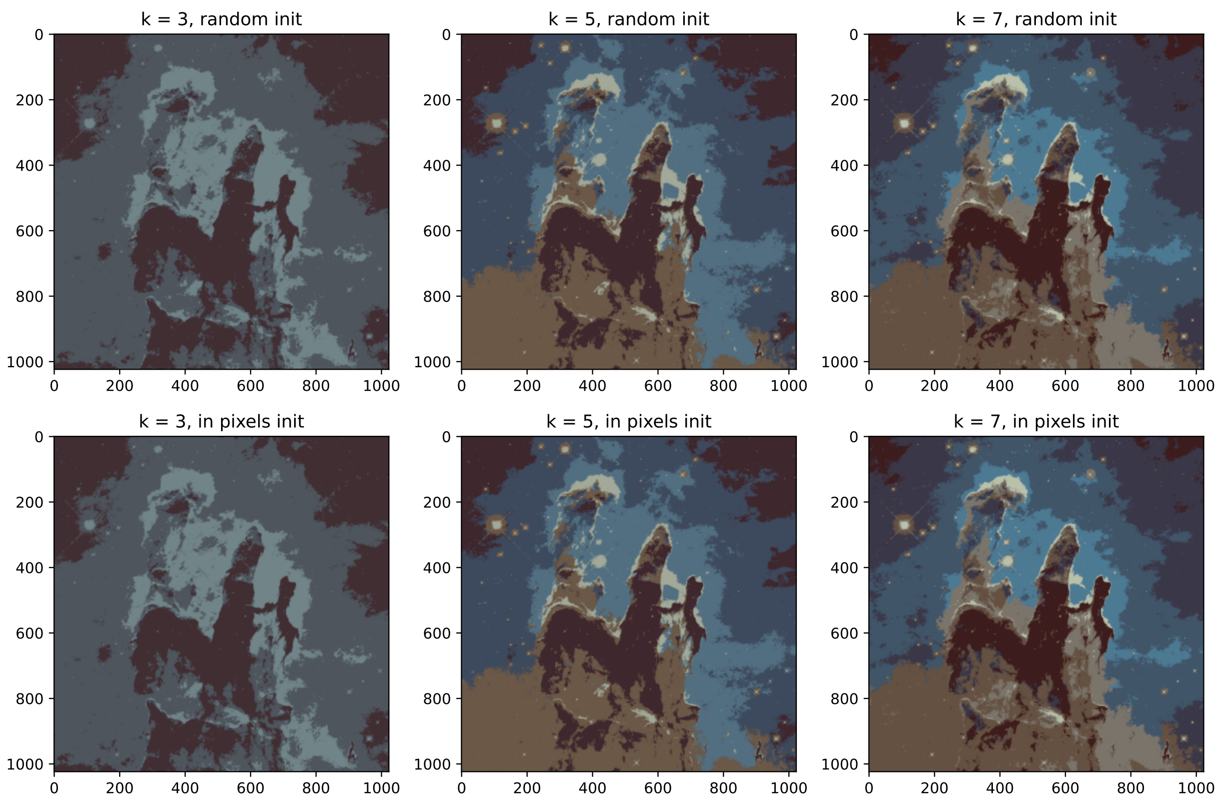

Figure 2. Results of k-means clustering on the *Pillars of Creation* with $k = 3, 5, 7$ along with comparisons between random initialization (top row) and initialization from original pixels (bottom row). All executions are bounded by an upper limit of 1000 computational loops

In Figure 2, the consistency in colour is clearly visible for both initialization modes. In previous runs, this is not always the case. I find that randomizing initial clusters is more prone to empty cluster since the full colour range is way larger than the colour range of the sample image. In some cases, nearly half of the initial centroids were discarded, which makes the result image quality undesirable.

The amount of colour in a picture contributes greatly in its level of detail. In samples with smaller k values, the detail and depth is greatly reduced, most noticeably in the background dust clouds and pillars. While the overall size for storing the image is reduced, it comes with a cost of lowering the image quality. However, I believe this algorithm would work nicely in extracting the primary colours of the image.

In Appendix A, I present some extra experimental runs with a variety of samples. Some instances of empty cluster are also visible in that section through the inconsistency between the number of colour present and k value.

ACKNOWLEDGEMENTS

I would like to thank the lecturers of the *Applied Mathematics and Statistics* course at the Ho Chi Minh University of Science (Nguyen Dinh Thuc, Tran Ha Son, Nguyen Van Quang Huy, and Ngo Dinh Hy) for their instruction in theory and lab session as well as support during the project duration.

The testing samples used in this project are products of NASA/ESA Hubble Space Telescope, Twitter user @hitsukuya, miHoYo Co. Ltd. and unknown creator of an Internet *meme*. All data were used for educational and research purposes.

The structure and style of this report is based on the Monthly Notices of the Royal Astronomical Society (MNRAS) template and instructions for authors (https://academic.oup.com/mnras/).

Software: python 3.10 (https://www.python.org/), numpy 1.25 (https://numpy.org/), matplotlib 3.6.0 (https://matplotlib.org/), pillow 9.2 (https://python-pillow.org/).

REFERENCES

Harris, C.R., Millman, K.J., van der Walt, S.J. et al. 2020, Nature, 585, 357–362
J. D. Hunter 2007, Computing in Science & Engineering, 9, 90-95
”*k*-means clustering”, Wikipedia
”K means Clustering – Introduction”, GeeksforGeeks
priyarajtt 2022, ”How to Convert images to NumPy array?”, GeeksforGeeks
riturajsaha 2021, ”Why is Numpy faster in Python?”, GeeksforGeeks
”Bài 4: K-means Clustering”, machinelearningcoban.com
Yue Harriet Huang 2015, ”How do I use np.newaxis?”, Stack Overflow
Joshua Ebner 2018, ”Numpy Axes, Explained”, Sharp Sight
NumPy Documentation on Broadcasting (2023), NumPy User Guide

APPENDIX A: K-MEANS COLOUR REDUCTION ON OTHER SAMPLE IMAGES

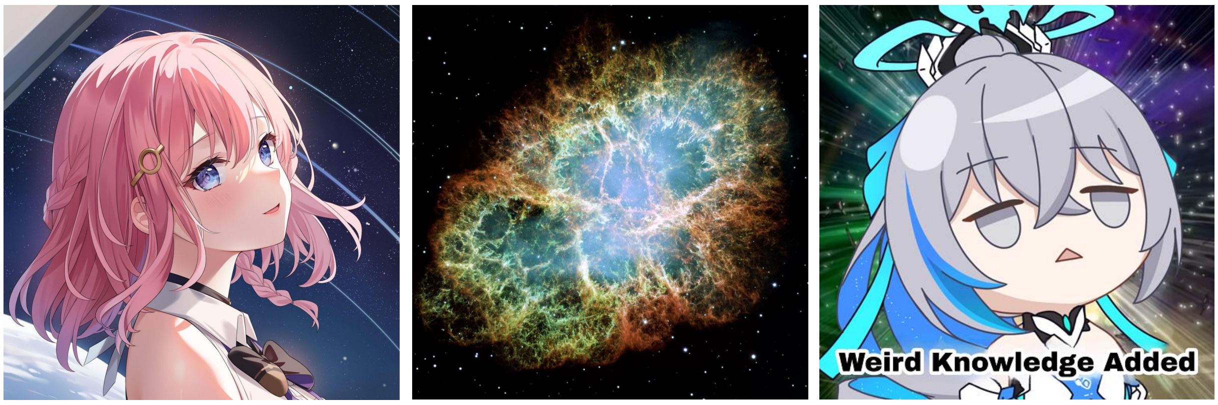

Figure A1. The original sample images. The resolution of each sample are (from left to right): 1080x1080 pixels, 1280x1280 pixels, 500x500 pixels.

I ran *k*-means on a different set of images with different colour ranges to observe its effect. Most plots in this section are not as consistent in terms of colour as the one illustrated in Fig. 2 due to the unpredictability of the initialization modes. The setting of each run is kept the same as in Section 3 with $k = 3, 5, 7$ and both initialization modes are applied on each value.

While I am aware of the impact of image resolution to the performance of the program as well as the overall quality, I will only focus on the colour aspect of the available images at the aforementioned resolution. Other executions on higher resolution image of the same object can result in different output.

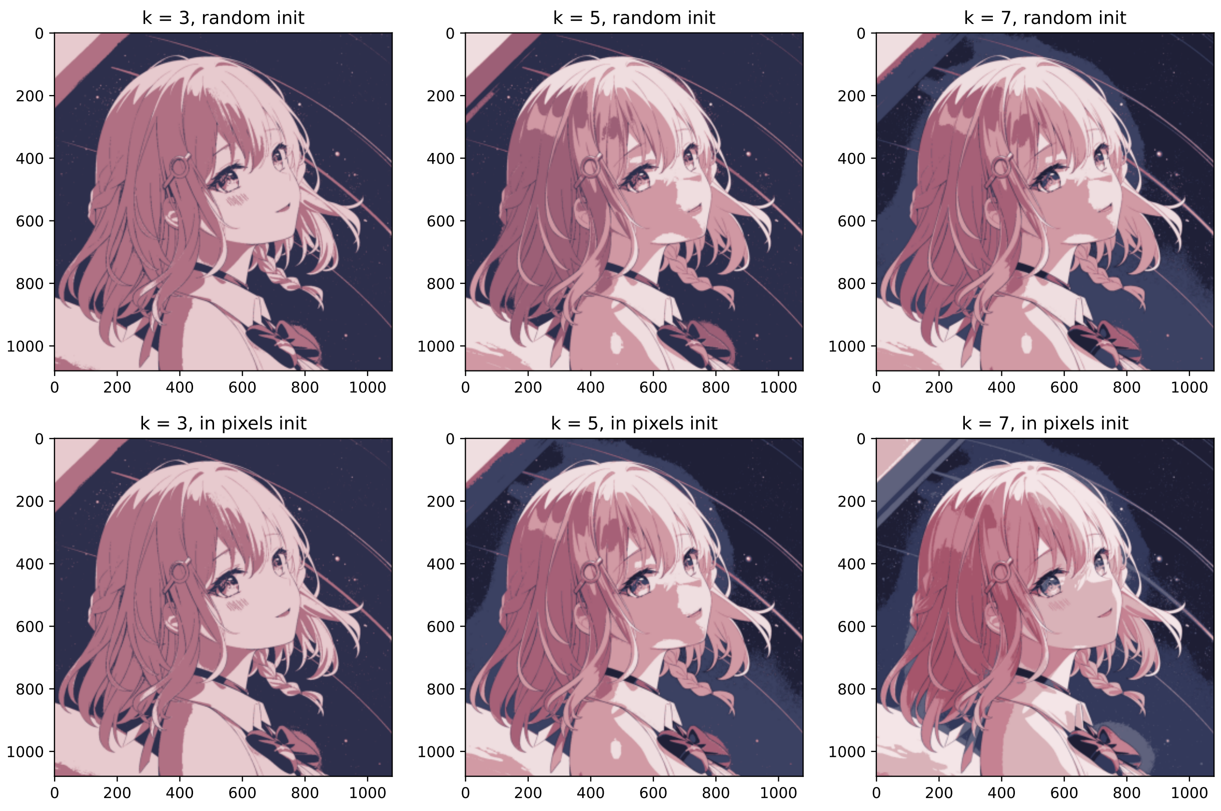

Figure A2. Same as Fig. 2 but for an image of a character by miHoYo Co. Ltd., drawn by Twitter user @hitsukuya. The effect of empty clusters are visible in random initialization through the image quality, however it is not present in in_pixels init mode.

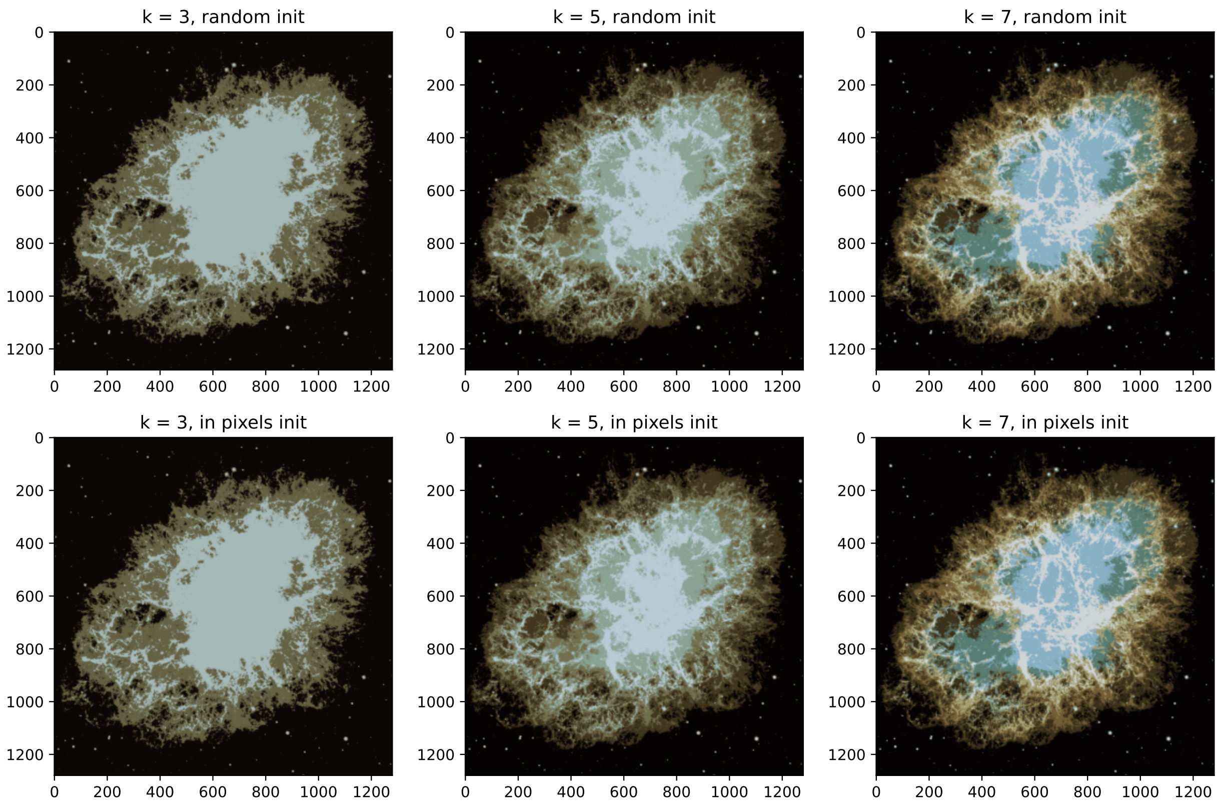

Figure A3. Same as Fig. 2 but for a photo of the Crab Nebula, taken by the Hubble Space Telescope. There are no empty clusters produced during the execution of this run.

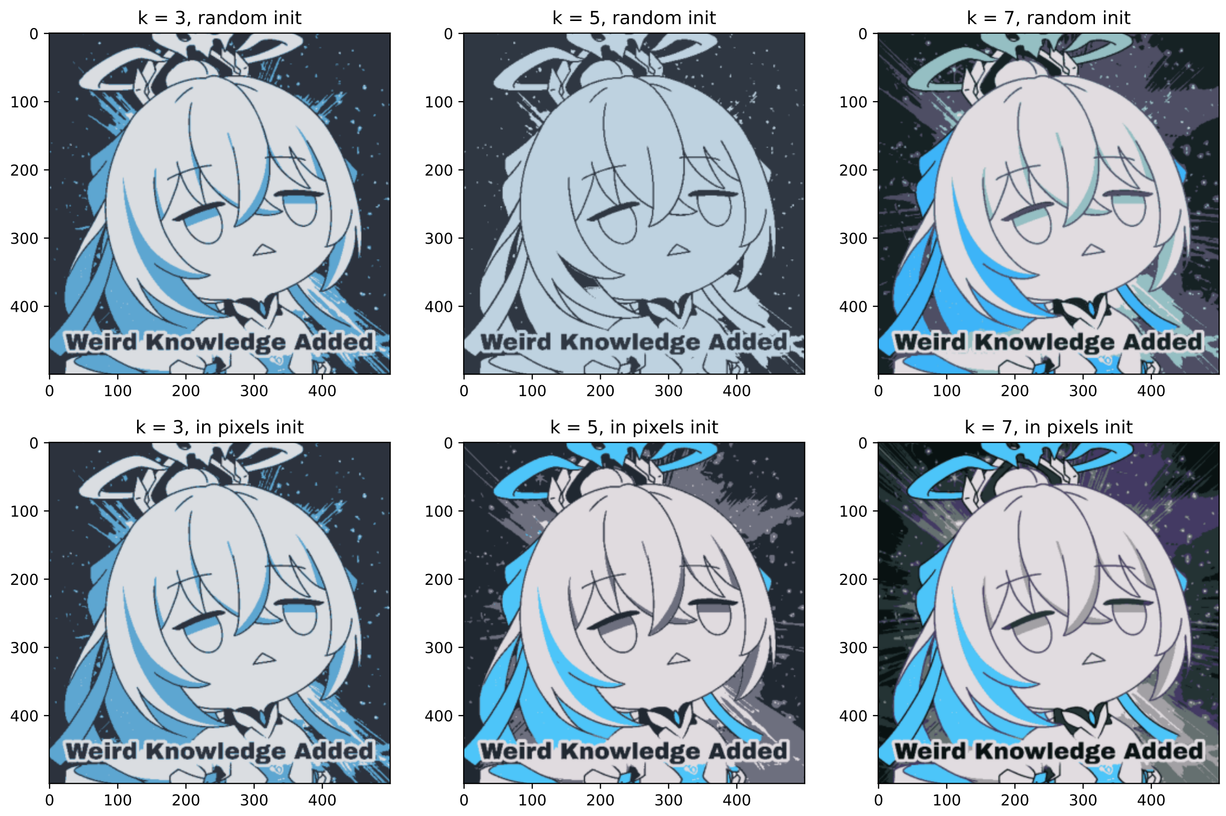

Figure A4. Same as Fig. 2 but for an image of a character by miHoYo Co. Ltd., made into an Internet *meme* by an unknown author. Effects of empty clusters are clearly visible for both initialization modes ran at larger k values.