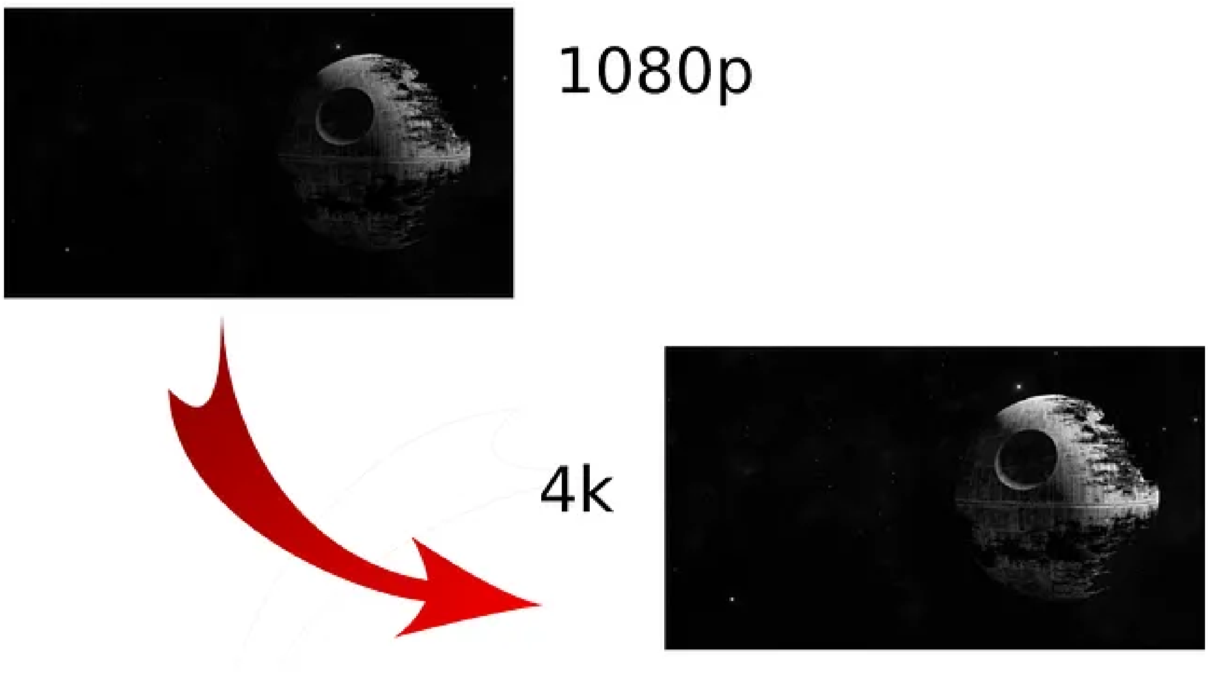
Image Upscaling using Bicubic Interpolation

Amanrao · Follow
7 min read · Sep 13

Listen Share

Image Upscaling is the process by which we increase the resolution of the image while minimizing the loss in image quality that occurs due to enlarging the image.

This article will show you how I implemented image upscaling using Bicubic interpolation in Python.



Before I explain how I implemented Bicubic interpolation here is a list of different traditional upscaling techniques:

Nearest Neighbor (NN):

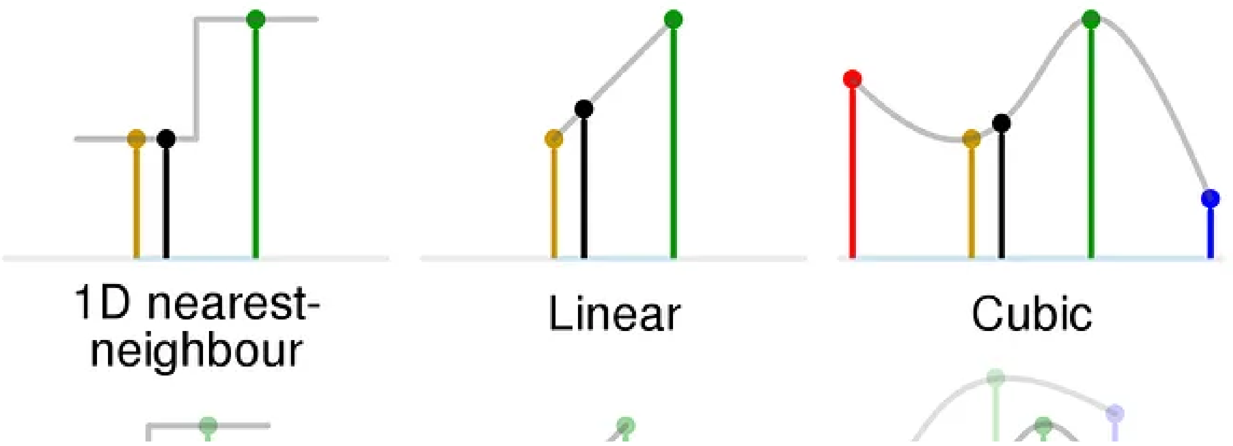
- Nearest Neighbor interpolation is the simplest upscaling method.
- Each pixel in the upscaled image is assigned the value of its nearest neighbor in the original image.
- It's fast but tends to produce blocky and pixelated results.

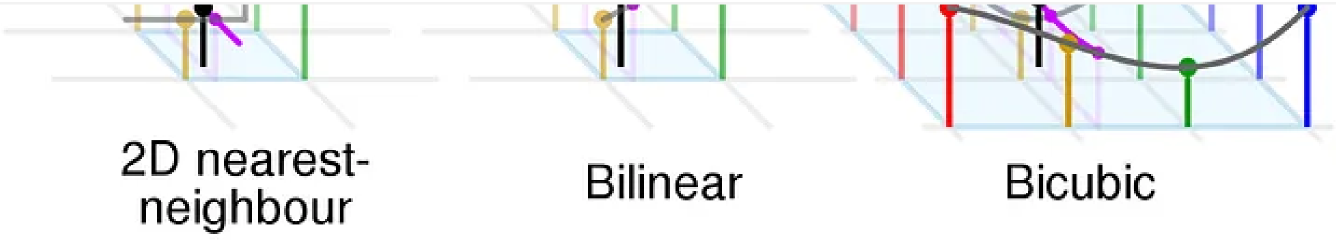
Bilinear Interpolation:

- Bilinear interpolation calculates a weighted average of the four nearest neighbors' pixel values in the original image.
- This method produces smoother results compared to NN but may still lack fine details.

Lanczos Resampling:

- Lanczos resampling is a high-quality upscaling method that uses a convolution filter with a window function to interpolate pixel values.
- It preserves sharpness and details but can be computationally intensive.





Bicubic VS Bilinear

Bilinear interpolates value of unknown / new pixels by assuming a linear change in channel value from pixel to pixel. Hence, we need to only find the distance or offset of the unknown pixel from the known pixels and plug it into the linear equation to find channel value for that pixel.

Bicubic, on the other hand, takes in to consideration the intensity values of more neighboring pixels to create a smoother curve by using the additional information to create non linear functions which reduce the occurrence of shortcomings produced by the bilinear method.

In 1d :

Bilinear uses **2** pixels to extrapolate pixel value
Bicubic uses **4** pixels to extrapolate pixel value

In 2d :

Bilinear uses **2 * 2 = 4** pixels to extrapolate pixel value
Bicubic uses **4 * 4 = 16** pixels to extrapolate pixel value

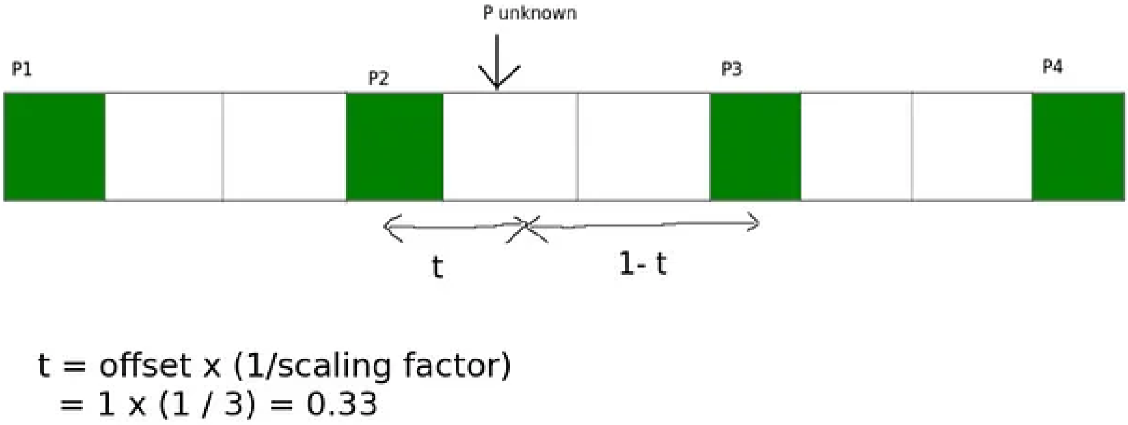
Hence, Bicubic is computationally more expensive than bilinear interpolation for upscaling images since it needs to compute more values as intermediate steps.

Formulae:

p = neighboring known points
q = basis function

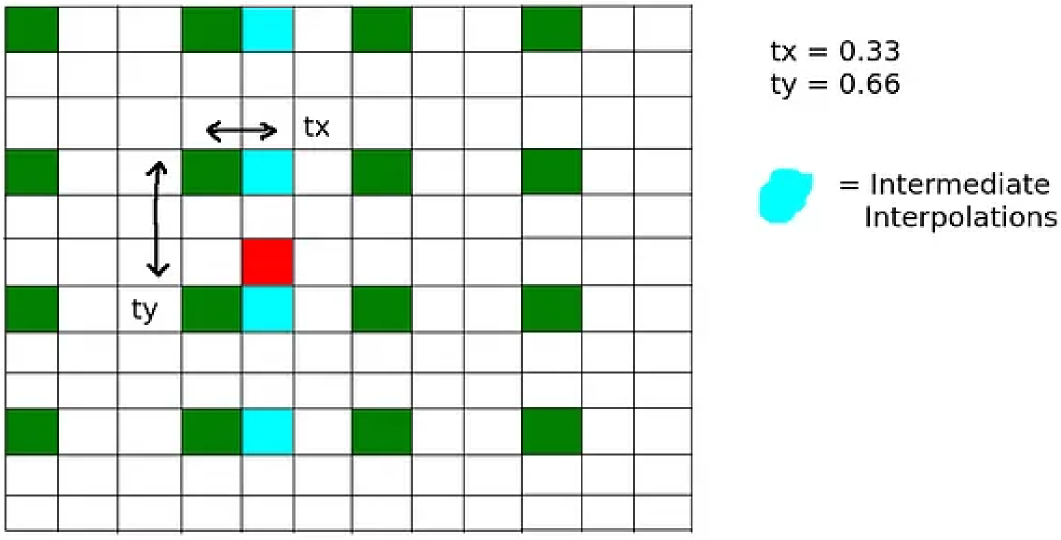
> **Bilinear:**
>
> p (unknown) = p1q1 + p2q2
>
> *where **p1 and p2 are value of surrounding known pixels***
>
> *and*
>
> ***q1 and q2 are offset of pixel from point p1 and p2 respectively***
> **normalized** between 0 and 1

> **such that** q1 + q2 = 1
>
> *Note:* ***q1*** *=* ***t*** *and* ***q2*** *=* ***1 — t*** *where t is the offset*



> **Bicubic:**
>
> *The formula for bicubic looks similar to Bilinear's:*
>
> p (unknown) = p1q1 + p2q2 + p3q3 + p4q4
>
> *where* ***p2*** *and* ***p3*** *are the points which are nearest to the unknown point and* ***p1*** *and* ***p4*** *are other points next to* ***p2*** *and* ***p3*** *respectively.*
>
> *The only thing that changes is the basis function*
>
> $q1 = (-t^3 + 2t^2 - t) / 2$
> $q2 = (3t^3 - 5t^2 + 2) / 2$
> $q3 = (-3t^3 + 4t + t) / 2$
> $q4 = (t^3 - t^2) / 2$

we may need to interpolate values of multiple points or pixels before finally interpolating the value of the pixel we desire.



Applying Bicubic Interpolation for Image

When applying Bicubic Interpolation on 2d images, we will observe the following cases:

1. The pixel lies on the same horizontal or vertical line as the known pixels
2. The pixel does not lie on the same horizontal or vertical line as the known pixels
3. Edge cases like corner and edge pixels for which we don't have enough info to interpolate it's values

Case 3 can be solved by applying either NN or Bilinear or other methods like clamping and wrapping (I used NN in my implementation for simplicity).

Case 1 can be solved by applying the formula only once, since we only have offset in the x or the y direction i.e. in a single dimension.

For Case 2 though, you will have to perform multiple interpolations (and / or keep a lookup table to optimize computation) in order to interpolate the value of the desired pixel.

My Implementation

I created a quick and dirty implementation of bicubic interpolation in python. My script takes in a 1080p picture and outputs a 4k one.

You can find it Here

Here is the algorithm for my script:

Importing the image

```
 image = cv2.imread(input_file)
```

Splitting image into R, G, B color channels

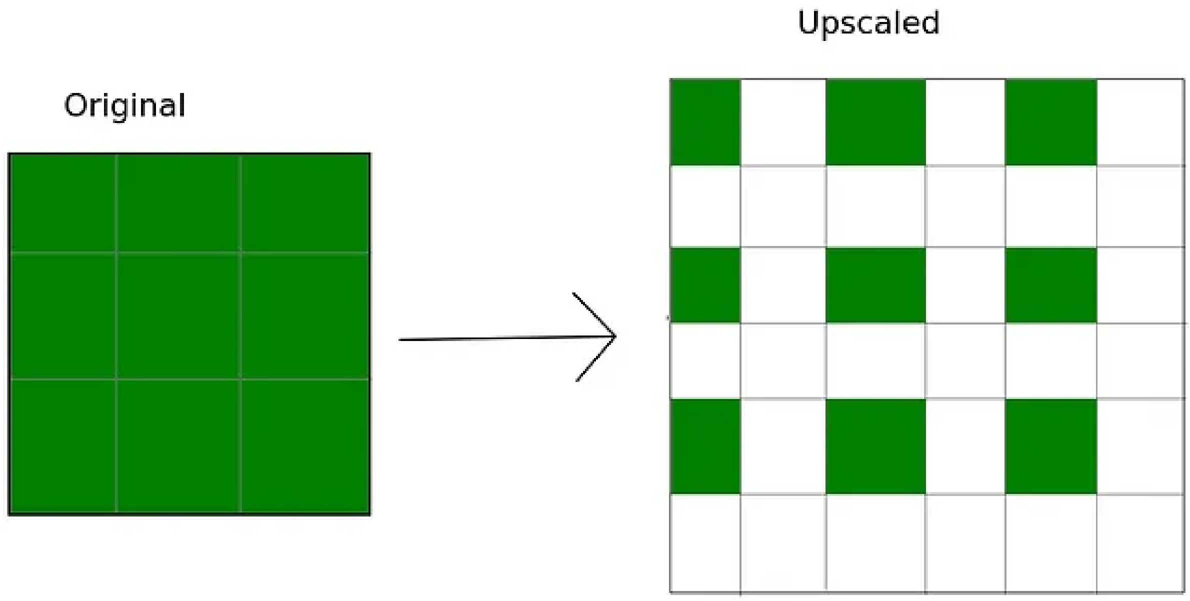
```
b, g, r = cv2.split(image)
```

Creating a new image which is scaled by 2 (1920 x 1080 multiplied by 2 is equal to 3840 x 2160 which is 4k)

```
r_upscale = np.zeros((int(height * 2), int(width * 2)))
g_upscale = np.zeros((int(height * 2), int(width * 2)))
b_upscale = np.zeros((int(height * 2), int(width * 2)))
```

Placing known pixels onto new image

```
for i in range(0, r_upscale.shape[0], 2):
 for j in range(0, r_upscale.shape[1], 2):
   r_upscale[i, j] = r[int(i/2), int(j/2)]
   g_upscale[i, j] = g[int(i/2), int(j/2)]
   b_upscale[i, j] = b[int(i/2), int(j/2)]
```

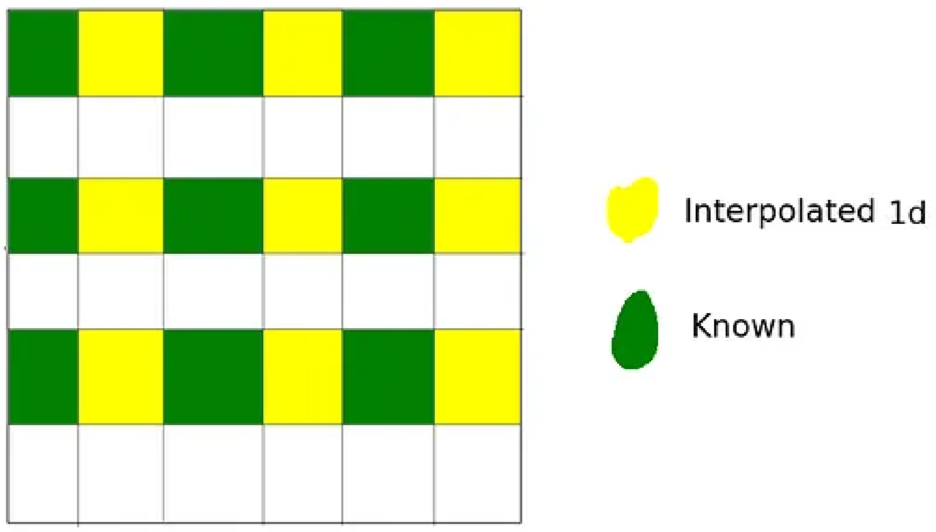


Placing pixels of the original image on the new, upscaled image

Interpolating pixels on the same horizontal line as known pixels

```
for i in range(0, r_upscale.shape[0], 2):
        for j in range(1, r_upscale.shape[1], 2):
            if j % 2 != 0:
                if j < 2 or j >= r_upscale.shape[1] - 3:
                    r_upscale[i, j] = r_upscale[i, j-1]
                    g_upscale[i, j] = g_upscale[i, j-1]
                    b_upscale[i, j] = b_upscale[i, j-1]

                else:
                    t = 0.5
                    tt = t**2
                    ttt = t**3

                    q1 = (-ttt + 2*tt - t) / 2
                    q2 = (3*ttt - 5*tt + 2) / 2
                    q3 = (-3*ttt + 4*t + t) / 2
                    q4 = (ttt - tt) / 2

                    p1 = r_upscale[i, j-3]
                    p2 = r_upscale[i, j-1]
                    p3 = r_upscale[i, j+1]
                    p4 = r_upscale[i, j+3]

                    r_upscale[i, j] = round(p1 * q1 + p2 * q2 + p3 * q3 + p4 *
```

```
                    p1 = g_upscale[i, j-3]
                    p2 = g_upscale[i, j-1]
                    p3 = g_upscale[i, j+1]
                    p4 = g_upscale[i, j+3]

                    g_upscale[i, j] = round(p1 * q1 + p2 * q2 + p3 * q3 + p4 *

                    p1 = b_upscale[i, j-3]
                    p2 = b_upscale[i, j-1]
                    p3 = b_upscale[i, j+1]
                    p4 = b_upscale[i, j+3]

                    b_upscale[i, j] = round(p1 * q1 + p2 * q2 + p3 * q3 + p4 *
```

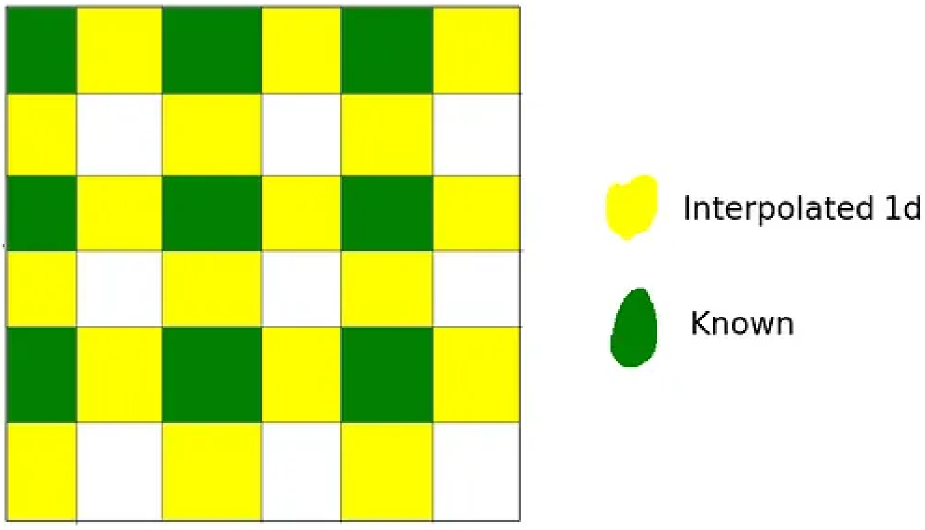


Placing pixels lying on the same horizontal direction as known pixel and applying 1d interpolation

Interpolating pixels on the same vertical line as known pixels

```
for j in range(0, r_upscale.shape[1], 2):
        for i in range(1, r_upscale.shape[0], 2):
            if i % 2 != 0:
                if i < 2 or i >= r_upscale.shape[0] - 3:
                    r_upscale[i, j] = r_upscale[i-1, j]
                    g_upscale[i, j] = g_upscale[i-1, j]
                    b_upscale[i, j] = b_upscale[i-1, j]
```

```
                else:
                    t = 0.5
                    tt = t**2
                    ttt = t**3

                    q1 = (-ttt + 2*tt - t) / 2
                    q2 = (3*ttt - 5*tt + 2) / 2
                    q3 = (-3*ttt + 4*t + t) / 2
                    q4 = (ttt - tt) / 2

                    p1 = r_upscale[i-3, j]
                    p2 = r_upscale[i-1, j]
                    p3 = r_upscale[i+1, j]
                    p4 = r_upscale[i+3, j]

                    r_upscale[i, j] = round(p1 * q1 + p2 * q2 + p3 * q3 + p4 *

                    p1 = g_upscale[i-3, j]
                    p2 = g_upscale[i-1, j]
                    p3 = g_upscale[i+1, j]
                    p4 = g_upscale[i+3, j]

                    g_upscale[i, j] = round(p1 * q1 + p2 * q2 + p3 * q3 + p4 *

                    p1 = b_upscale[i-3, j]
                    p2 = b_upscale[i-1, j]
                    p3 = b_upscale[i+1, j]
                    p4 = b_upscale[i+3, j]

                    b_upscale[i, j] = round(p1 * q1 + p2 * q2 + p3 * q3 + p4 *
```

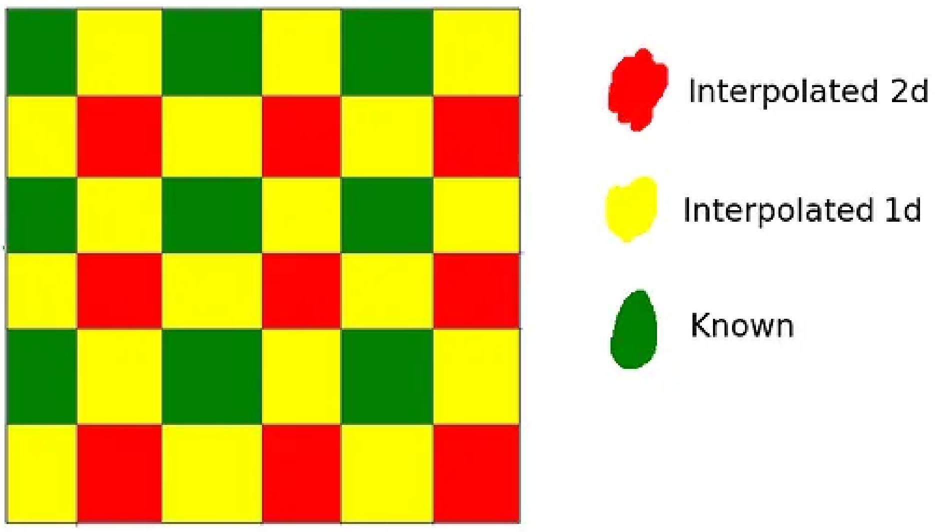


Placing pixels lying on the same vertical direction as known pixel and applying 1d interpolation

Interpolating rest of the pixels with previously interpolated pixels

```
for i in range(1, r_upscale.shape[0], 2):
        for j in range(1, r_upscale.shape[1], 2):
            if j<3 or j>r_upscale.shape[1]-5 or i<3 or i>r_upscale.shape[0]-5:
                r_upscale[i, j] = r_upscale[i-1, j]
                g_upscale[i, j] = g_upscale[i-1, j]
                b_upscale[i, j] = b_upscale[i-1, j]

            else:
                t = 0.5
                tt = t**2
                ttt = t**3

                q1 = (-ttt + 2*tt - t) / 2
                q2 = (3*ttt - 5*tt + 2) / 2
                q3 = (-3*ttt + 4*t + t) / 2
                q4 = (ttt - tt) / 2

                p1 = r_upscale[i-3, j]
                p2 = r_upscale[i-1, j]
                p3 = r_upscale[i+1, j]
                p4 = r_upscale[i+3, j]

                r_upscale[i, j] = round(p1 * q1 + p2 * q2 + p3 * q3 + p4 * q4)

                p1 = g_upscale[i-3, j]
```

```
            p2 = g_upscale[i-1, j]
            p3 = g_upscale[i+1, j]
            p4 = g_upscale[i+3, j]

            g_upscale[i, j] = round(p1 * q1 + p2 * q2 + p3 * q3 + p4 * q4)

            p1 = b_upscale[i-3, j]
            p2 = b_upscale[i-1, j]
            p3 = b_upscale[i+1, j]
            p4 = b_upscale[i+3, j]

            b_upscale[i, j] = round(p1 * q1 + p2 * q2 + p3 * q3 + p4 * q4)
```



Placing rest of the pixels and interpolating their value using 2d bicubic interpolation

Merging the 3 color channels into 1.

```
frame_upscale = cv2.merge([b_upscale, g_upscale, r_upscale])
```

Outputting new image

```
cv2.imwrite(output_file, frame_upscale)
```

Conclusion and Improvements To Be Made

The code I have written does have a lot which can be be improved. Designing it to take in images of any dimension as well as allowing it to be scaled as per the user's choice is one. The other being optimizing the above code by allowing for parallelization as well as DRY compliance. But this did help me uncover a layer of abstraction in how images are traditionally upscaled.

Hope you enjoyed this article! Connect with me on LinkedIn

Aman Out.

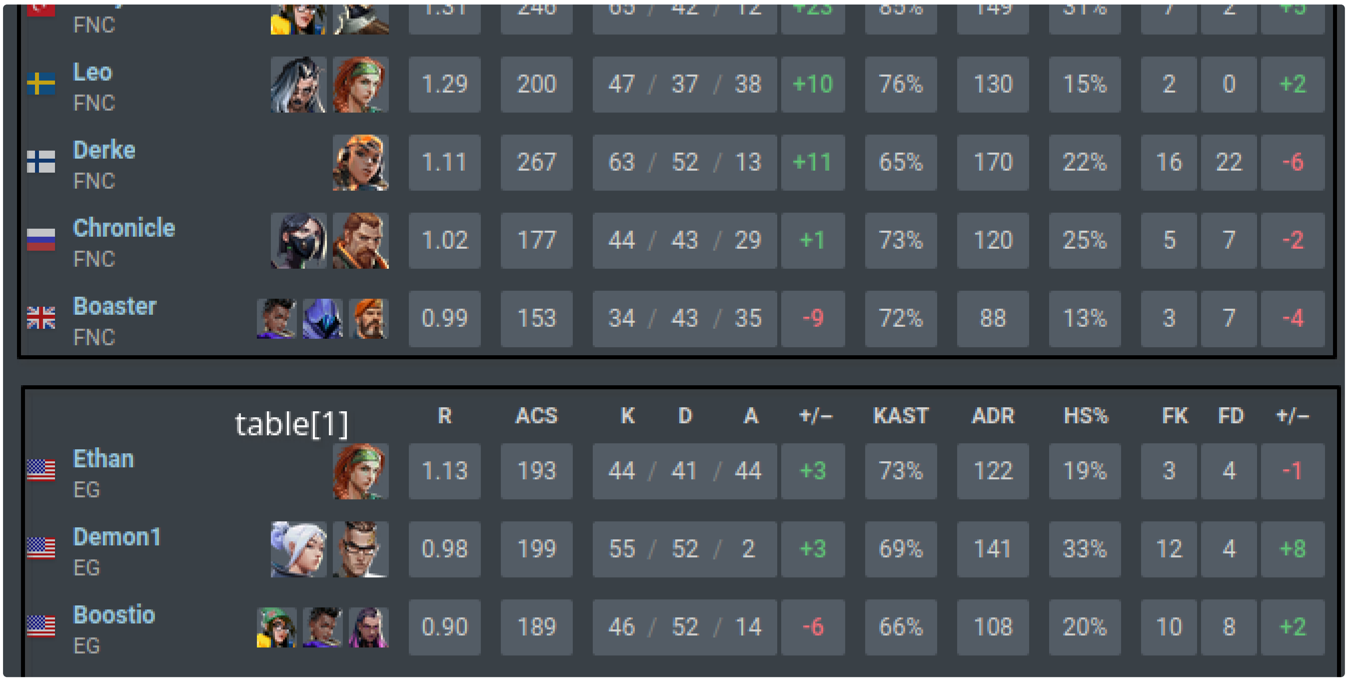
Data Science Image Processing Upscale Image Python Interpolation

Follow

Written by Amanrao

1 Follower

More from Amanrao



	R	ACS	K / D / A	+/–	KAST	ADR	HS%	FK	FD	+/–
FNC	1.31	246	65 / 42 / 12	+23	85%	149	31%	7	2	+5
Leo FNC	1.29	200	47 / 37 / 38	+10	76%	130	15%	2	0	+2
Derke FNC	1.11	267	63 / 52 / 13	+11	65%	170	22%	16	22	-6
Chronicle FNC	1.02	177	44 / 43 / 29	+1	73%	120	25%	5	7	-2
Boaster FNC	0.99	153	34 / 43 / 35	-9	72%	88	13%	3	7	-4

table[1]

	R	ACS	K / D / A	+/–	KAST	ADR	HS%	FK	FD	+/–
Ethan EG	1.13	193	44 / 41 / 44	+3	73%	122	19%	3	4	-1
Demon1 EG	0.98	199	55 / 52 / 2	+3	69%	141	33%	12	4	+8
Boostio EG	0.90	189	46 / 52 / 14	-6	66%	108	20%	10	8	+2

Amanrao

Creating A Valorant Player Stats Dataset

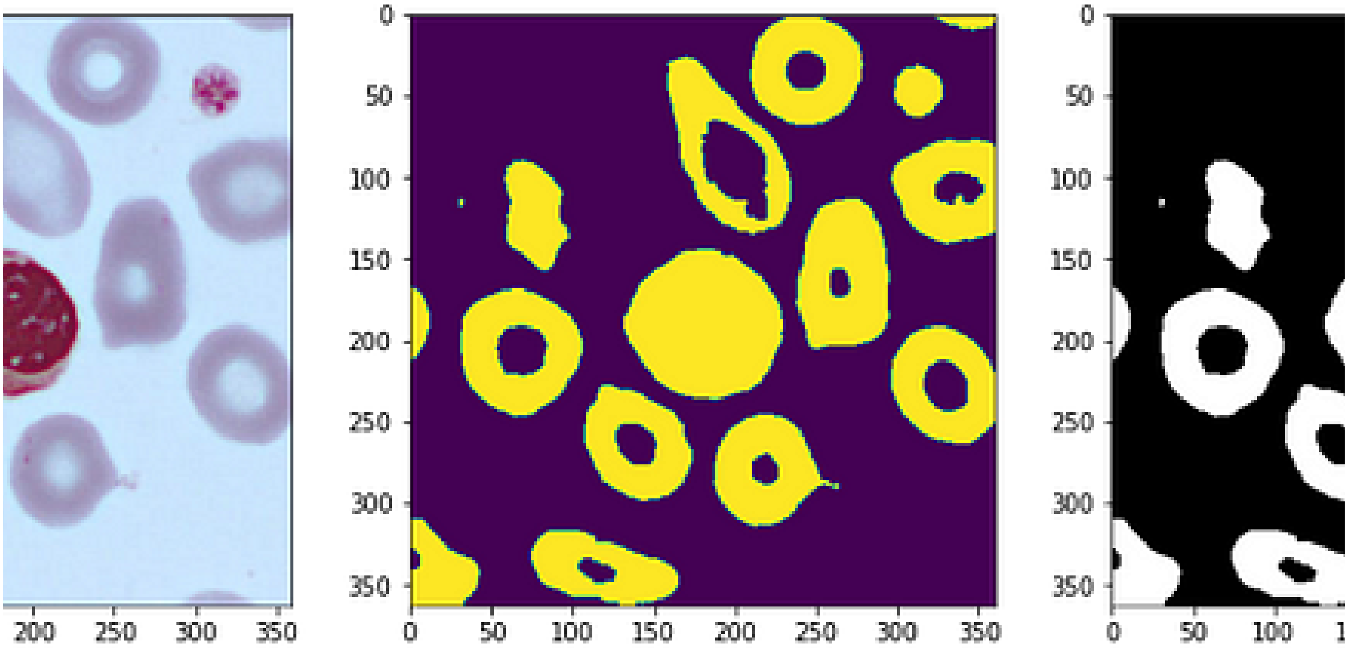
I thoroughly enjoyed Valorant Master Tokyo. From EDG's dominance against what were considered to be favorites in the tourney to EG's...

5 min read · Jul 17

2

See all from Amanrao

Recommended from Medium



Sasani S. Perera

OpenCV: Object Masking

Object masking is very important in Image processing. This article discusses the theories behind object masking and a basic demonstration.

6 min read · Jul 13

10

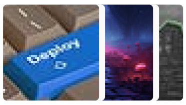
PyPose in PyTorch

PyPose: A Library for Robot Learning with Physics-based Optimization

PyPose is now part of the PyTorch Ecosystem!

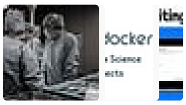
3 min read · Dec 6

Lists



Predictive Modeling w/ Python

20 stories · 685 saves

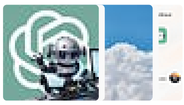


Coding & Development

11 stories · 311 saves

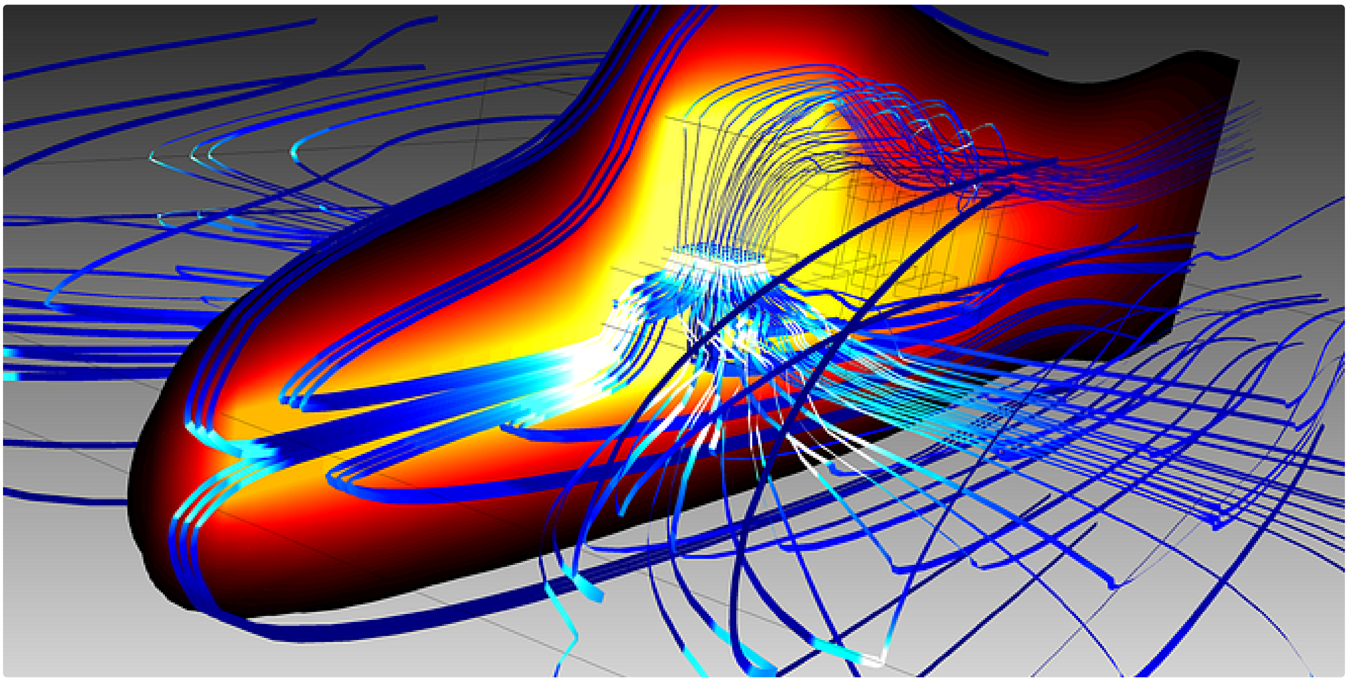
Practical Guides to Machine Learning

10 stories · 782 saves



ChatGPT

23 stories · 311 saves



Antonio Pappaterra

Python in Action: Simulating Fluid Dynamics and Structural Analysis with CFD and FEA

In the ever-evolving landscape of scientific and engineering simulations, Python has emerged as a powerhouse for researchers and engineers...

3 min read · Oct 18

10

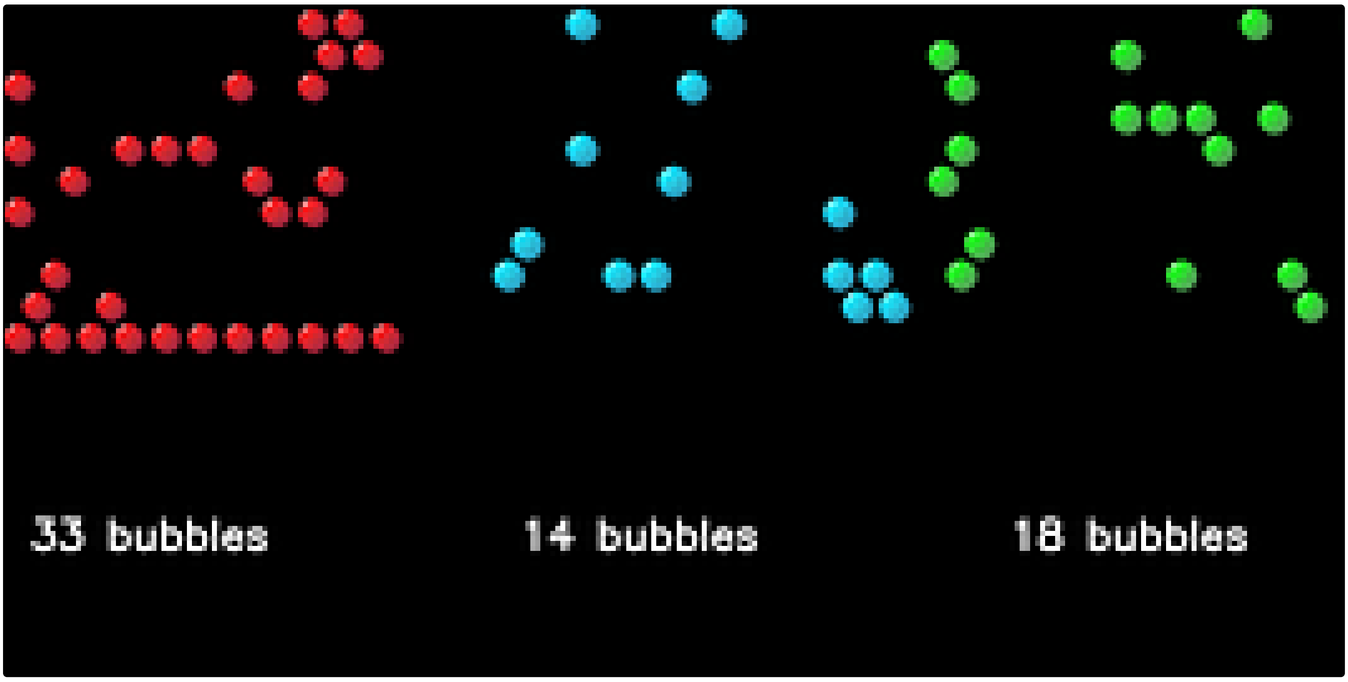
Mohapatra Abhilash in Vytah—future of space

What is Python being used for in space?

Somewhere, something incredible is waiting to be known -Sharon Begley

7 min read · 5 days ago

1



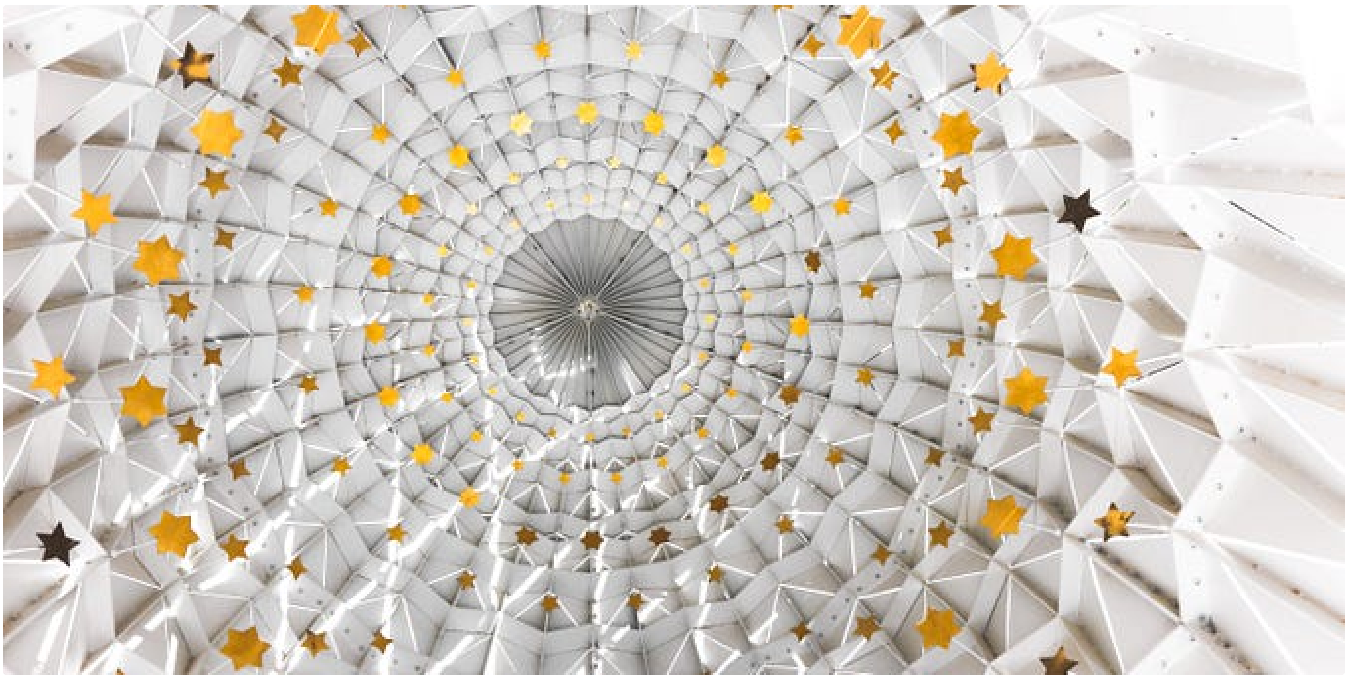
Lihi Gur Arie, PhD in Towards Data Science

Image color Segmentation by K-means clustering algorithm

Guided tutorial explaining how to train a simple U-net for semantic segmentation of microorganisms dataset

6 min read · Dec 6, 2022

268



Okan Yenigün in AWS Tip

Corner Detection in Image Processing

Implementation in Python-OpenCV

7 min read · Dec 3

10

See more recommendations