Image Segmentation Using Color Spaces in OpenCV + Python

by Rebecca Stone 26 Comments intermediate machine-learning

Tweet Share Email

Table of Contents

It may be the era of deep learning and big data, where complex algorithms analyze images by being shown millions of them, but color spaces are still surprisingly useful for image analysis. Simple methods can still be powerful.

In this article, you will learn how to simply segment an object from an image based on color in Python using OpenCV. A popular computer vision library written in C/C++ with bindings for Python, OpenCV provides easy ways of manipulating color spaces.

While you don't need to be already familiar with OpenCV or the other helper packages used in this article, it is assumed that you have at least a basic understanding of coding in Python.

What Are Color Spaces?

In the most common color space, RGB (Red
components. In more technical terms, RGB
take a value between 0 and 255, where the t

RGB is considered an "additive" color spa
of red, blue, and green light onto a black ba

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```


Here are a few more examples of colors in RGB:

Color	RGB value
Red	255, 0, 0
Orange	255, 128, 0
Pink	255, 153, 255

RGB is one of the five major color space models, each of which has many offshoots. **There are so many color spaces because different color spaces are useful for different purposes**.

In the printing world, **CMYK** is useful because it describes the color combinations required to produce a color from a white background. While the 0 tuple in RGB is black, in CMYK the 0 tuple is white. Our printers contain ink canisters of cyan, magenta, yellow, and black.

In certain types of medical fields, glass slides mounted with stained tissue samples are scanned and saved as images. They can be analyzed in **HED** space, a representation of the saturations of the stain types—hematoxylin, eosin, and DAB—applied to the original tissue.

HSV and HSL are descriptions of hue, saturation, and brightness/luminance, which are particularly useful for identifying contrast in images. These color spaces are frequently used in color selection tools in software and for web design.

In reality, color is a continuous phenomenon, meaning that there are an infinite number of colors. Color spaces, however, represent color through discrete structures (a fixed number of whole number integer values), which is acceptable since the human eye and perception are also limited. Color spaces are fully able to represent all the colors we are able to distinguish between.

Now that we understand the concept of color spaces, we can go on to use them in OpenCV.

Simple Segmentation Using Color Spaces

To demonstrate the color space segmentation technique, we've provided a small dataset of images of clownfish in the Real Python materials repository here for you to download and play with. Clownfish are easily identifiable by their bright orange color, so they're a good candidate for segmentation. Let's see how well we can find Nemo in an image.

The key Python packages you'll need to follow along are NumPy, the foremost package for scientific computing in Python, Matplotlib, a plotting library, and of course OpenCV. This articles uses OpenCV 3.2.0, NumPy 1.12.1, and Matplotlib 2.0.2. Slightly different versions won't make a significant difference in terms of following along and grasping the concepts.

If you are not familiar with NumPy or Matplotlib, you can read about them in the official NumPy guide and Brad Solomon's excellent article on Matplotlib.

Color Spaces and Reading Images in OpenCV

First, you will need to set up your environment. This article will assume you have Python 3.x installed on your system. Note that while the current version of OpenCV is 3.x, the name of the package to import is still `cv2`:

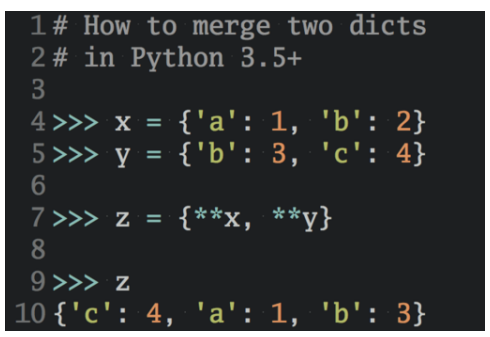
Python >>>

```
>>> import cv2
```

If you haven't previously installed OpenCV c
user-friendly tutorial for installing on differe
Once you've successfully imported OpenCV,
can save them all into a variable:

Python >>>

```
>>> flags = [i for i in dir(cv2) if i.s
```



The list and number of flags may vary slightly depending on your version of OpenCV, but regardless, there will be a lot! See how many flags you have available:

Python >>>

```
>>> len(flags)
258
>>> flags[40]
'COLOR_BGR2RGB'
```

The first characters after `COLOR_` indicate the origin color space, and the characters after the `2` are the target color space. This flag represents a conversion from BGR (Blue, Green, Red) to RGB. As you can see, the two color spaces are very similar, with only the first and last channels swapped.

You will need `matplotlib.pyplot` for viewing the images, and NumPy for some image manipulation. If you do not already have Matplotlib or NumPy installed, you will need to `pip3 install matplotlib` and `pip3 install numpy` before attempting the imports:

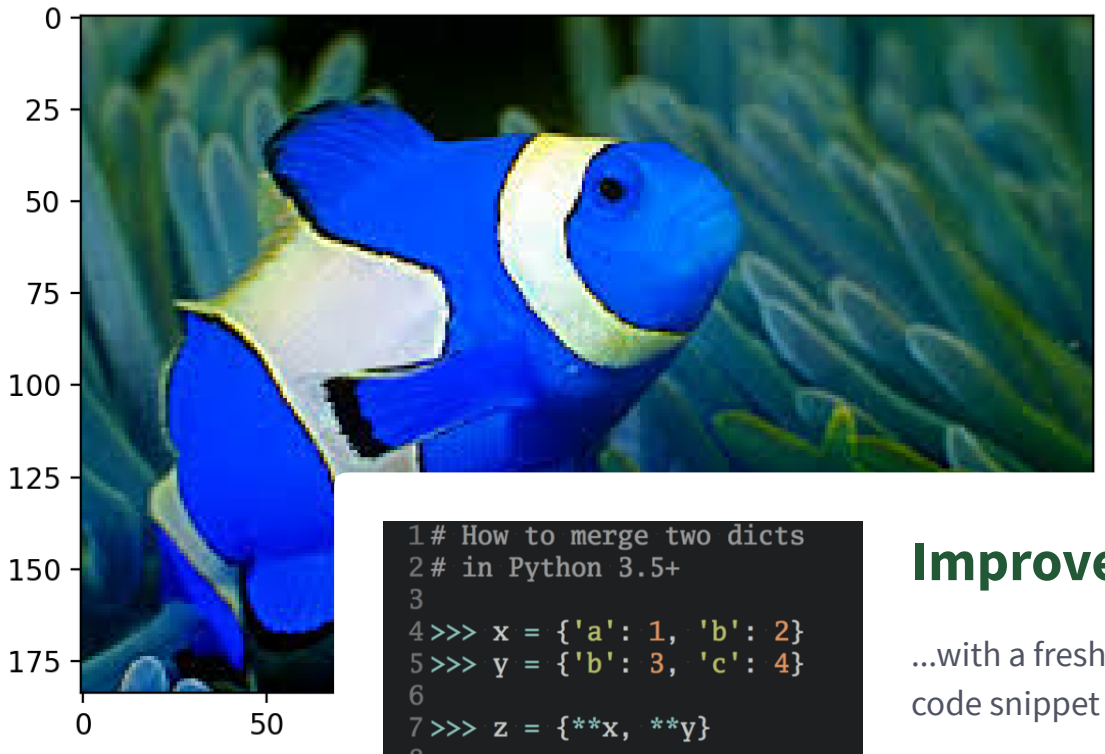
Python >>>

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
```

Now you are ready to load and examine an image. Note that if you are working from the command line or terminal, your images will appear in a pop-up window. If you are working in a Jupyter notebook or something similar, they will simply be displayed below. Regardless of your setup, you should see the image generated by the `show()` command:

Python >>>

```
>>> nemo = cv2.imread('./images/nemo0.jpg')
>>> plt.imshow(nemo)
>>> plt.show()
```



Hey, Nemo…or Dory? You'll notice that it lo
default reads images in BGR format. You c
this:

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```


Python

```
>>> nemo = cv2.cvtColor(nemo, cv2.COLOR_BGR2RGB)
>>> plt.imshow(nemo)
>>> plt.show()
```

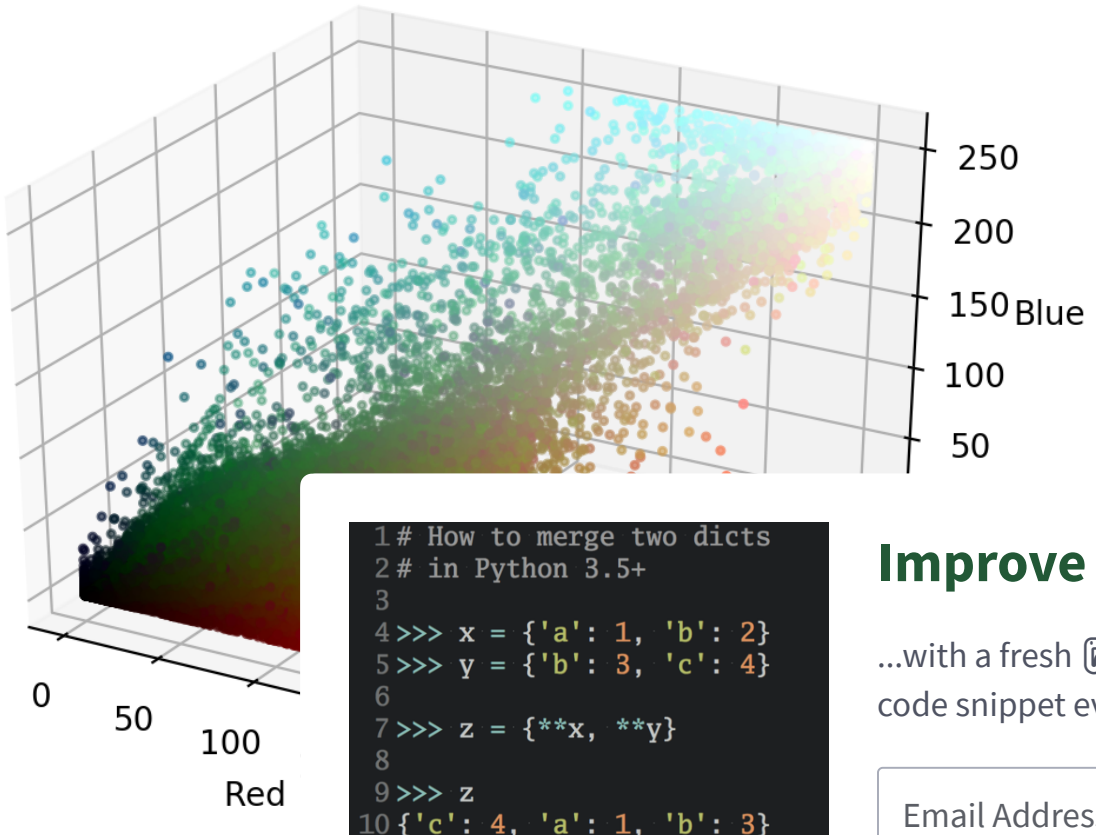
Now Nemo looks much more like himself.

Visualizing Nemo in RGB Color Space

HSV is a good choice of color space for segmenting by color, but to see why, let's compare the image in both RGB and HSV color spaces by visualizing the color distribution of its pixels. A 3D plot shows this quite nicely, with each axis representing one of the channels in the color space. If you want to know how to make a 3D plot, view the collapsed section:

How to Make a Colored 3D Scatter Plot Show/Hide

Here is the colored scatter plot for the Nemo image in RGB:



From this plot, you can see that the orange
blue values. Since parts of Nemo stretch ov
RGB values would not be easy.

Visualizing Nemo in HSV Color Space

We saw Nemo in RGB space, so now let's view him in HSV space and compare.

As mentioned briefly above, **HSV stands for Hue, Saturation, and Value (or brightness)**, and is a cylindrical color space. The colors, or hues, are modeled as an angular dimension rotating around a central, vertical axis, which represents the value channel. Values go from dark (0 at the bottom) to light at the top. The third axis, saturation, defines the shades of hue from least saturated, at the vertical axis, to most saturated furthest away from the center:

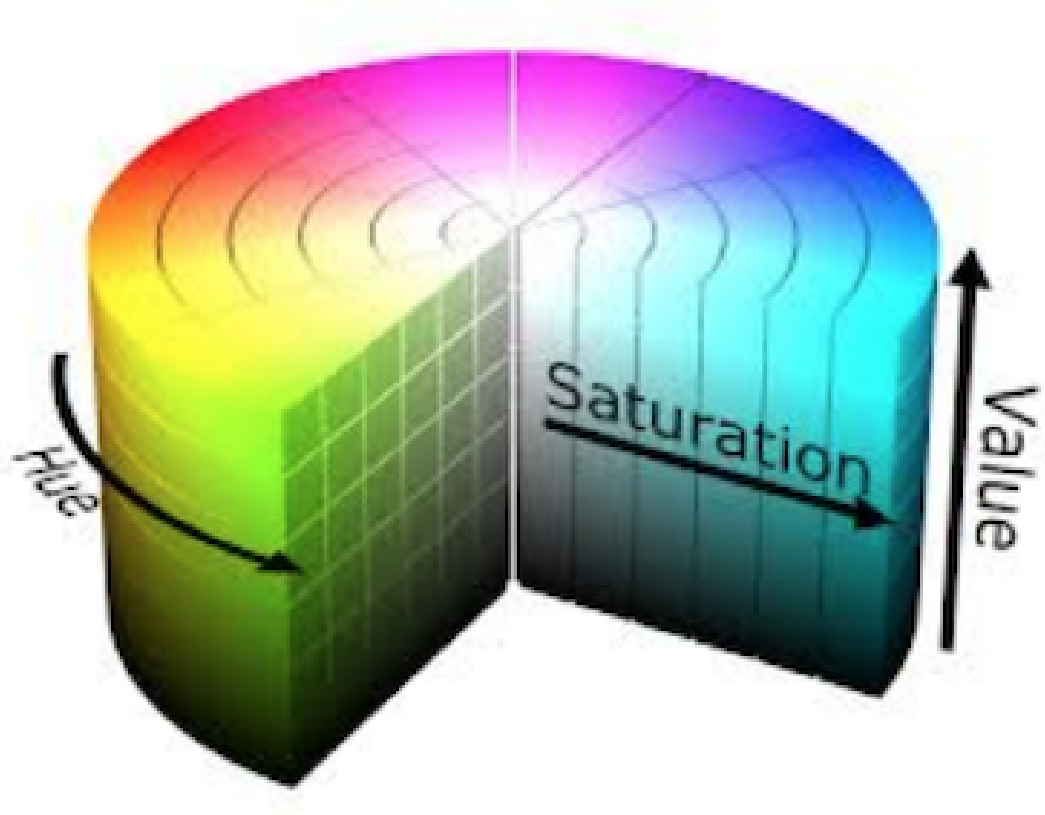


Image: Wikipedia

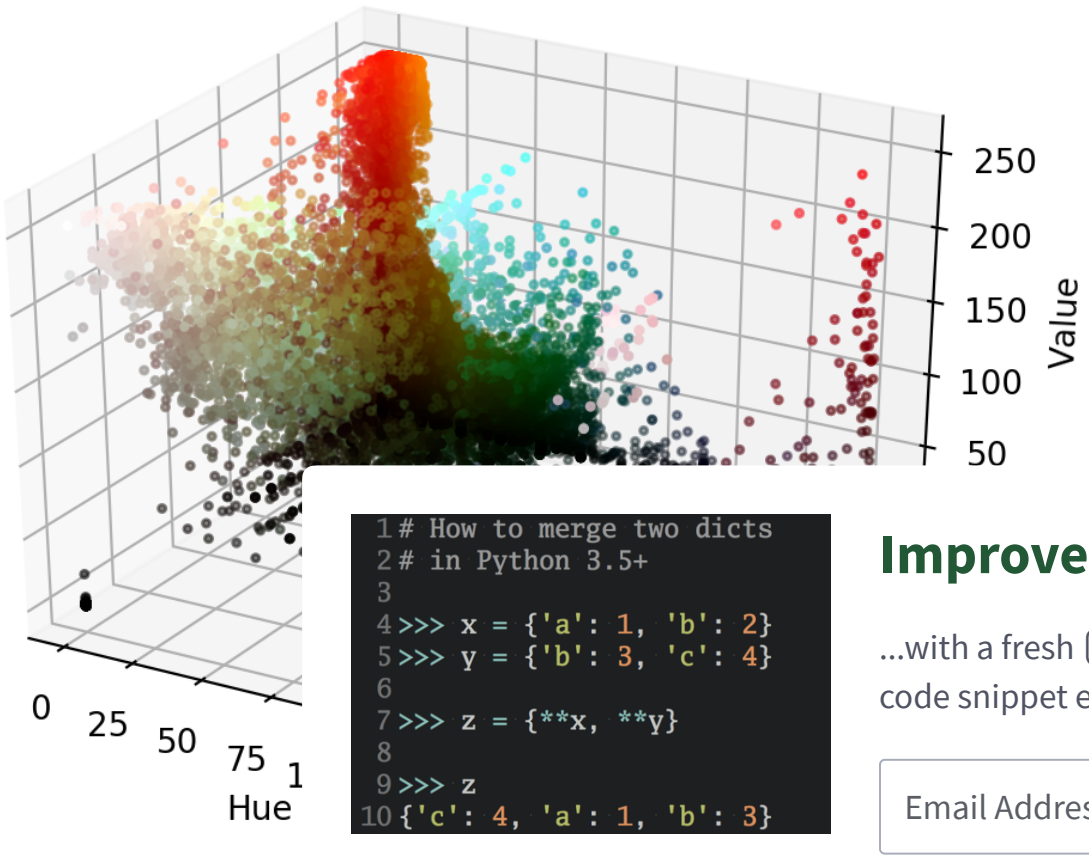
To convert an image from RGB to HSV, you can use `cvtColor()`:

Python >>>

```
>>> hsv_nemo = cv2.cvtColor(nemo, cv2.COLOR_RGB2HSV)
```

Now `hsv_nemo` stores the representation of Nemo in HSV. Using the same technique as above, we can look at a plot of the image in HSV, generated by the collapsed section below:

Generating the Colored 3D Scatter Plot for the Image in HSV Show/Hide



In HSV space, Nemo's oranges are much mo
oranges do vary, but they are mostly locate
be leveraged for segmentation.

Picking Out a Range

Let's threshold Nemo just based on a simple range of oranges. You can choose the range by eyeballing the plot above or using a color picking app online such as this RGB to HSV tool. The swatches chosen here are a light orange and a darker orange that is almost red:

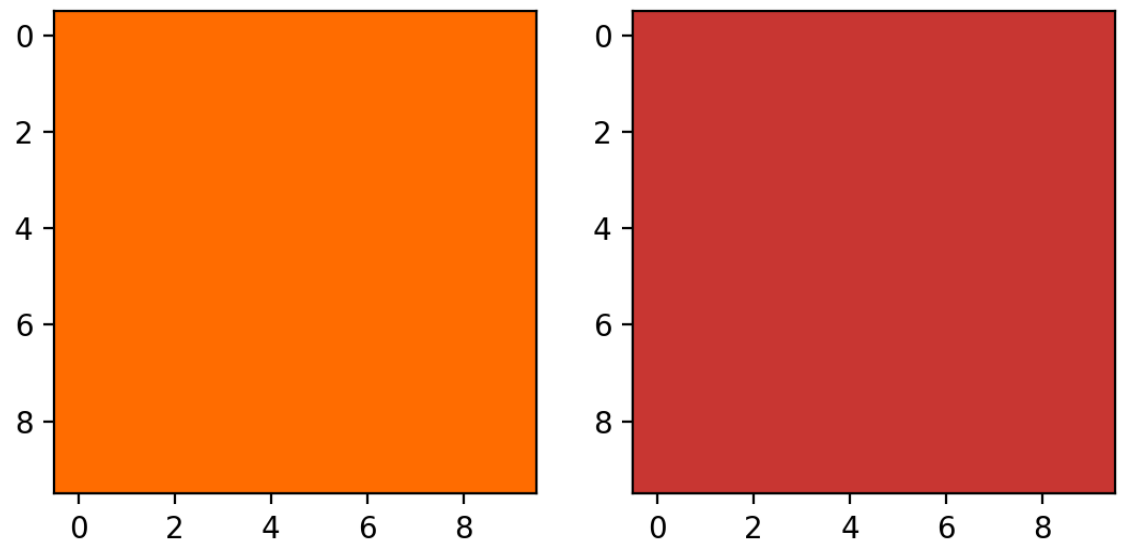
Python >>>

```
>>> light_orange = (1, 190, 200)
>>> dark_orange = (18, 255, 255)
```

If you want to use Python to display the colors you chose, click on the collapsed section:

Displaying the HSV Colors Chosen Show/Hide

That produces these images, filled with the chosen colors:



Once you get a decent color range, you can use `cv2.inRange()` to try to threshold Nemo. `inRange()` takes three parameters: **the image, the lower range, and the higher range**. It returns a binary mask (an `ndarray` of 1s and 0s) the size of the image where values of `1` indicate values within the range, and zero values indicate values outside:

Python >>>

```
>>> mask = cv2.inRange(hsv_nemo, light_orange, dark_orange)
```

To impose the mask on top of the original image, you can use `cv2.bitwise_and()`, which keeps every pixel in the given image if the corresponding value in the mask is `1`:

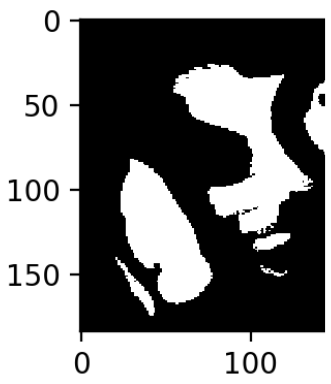
Python

```
>>> result = cv2.bitwise_and(nemo, nemo, mask=mask)
```

To see what that did exactly, let's view both the mask and the original image with the mask on top:

Python >>>

```
>>> plt.subplot(1, 2, 1)
>>> plt.imshow(mask, cmap="gray")
>>> plt.subplot(1, 2, 2)
>>> plt.imshow(result)
>>> plt.show()
```

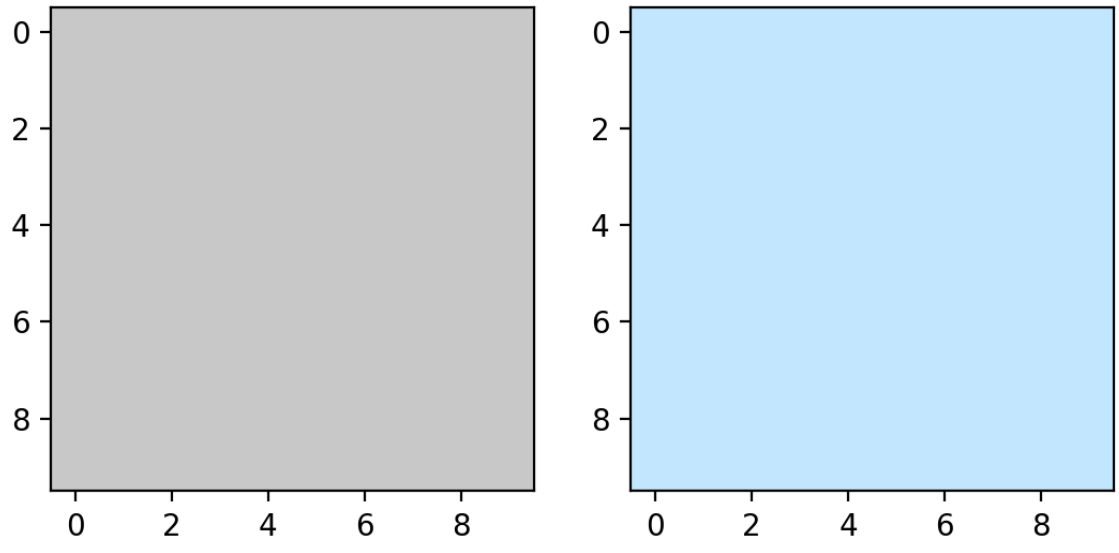


There you have it! This has already done a d
that Nemo also has white stripes… Fortunately, adding a second mask that looks for whites is very similar to what you did already with the oranges:

Python >>>

```
>>> light_white = (0, 0, 200)
>>> dark_white = (145, 60, 255)
```

Once you've specified a color range, you can look at the colors you've chosen:



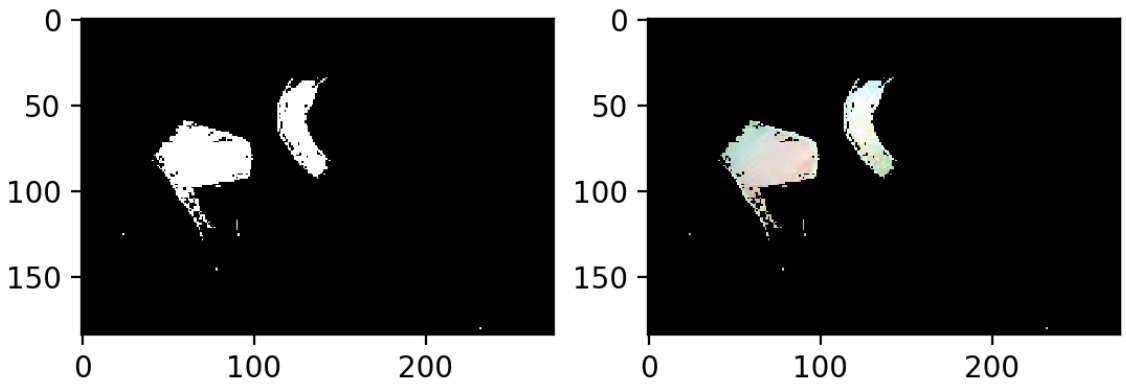
Displaying the Whites Show/Hide

The upper range I've chosen here is a very blue white, because the white does have tinges of blue in the shadows. Let's create a second mask and see if it captures Nemo's stripes. You can build a second mask the same way as you did the first:

Python >>>

```
>>> mask_white = cv2.inRange(hsv_nemo, light_white, dark_white)
>>> result_white = cv2.bitwise_and(nemo, nemo, mask=mask_white)

>>> plt.subplot(1, 2, 1)
>>> plt.imshow(mask_white, cmap="gray")
>>> plt.subplot(1, 2, 2)
>>> plt.imshow(result_white)
>>> plt.show()
```

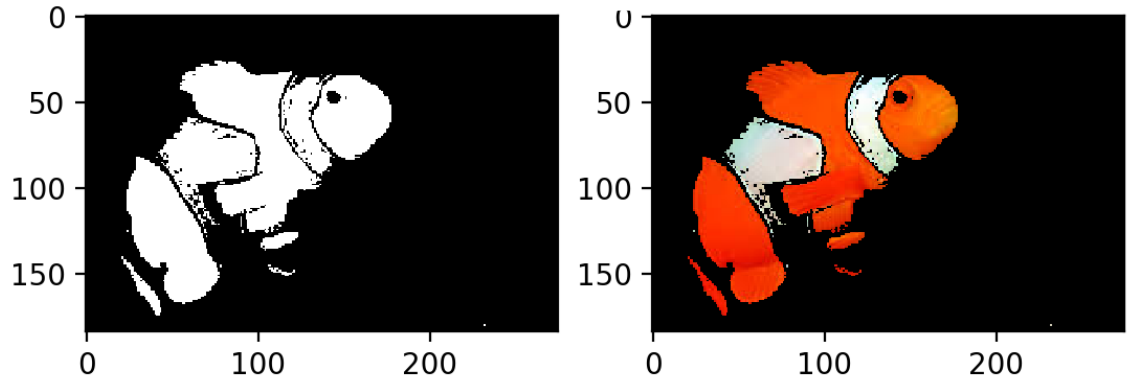


Not bad! Now you can combine the masks. Adding the two masks together results in 1 values wherever there is orange *or* white, which is exactly what is needed. Le

Python

```
>>> final_mask = mask + mask_white

>>> final_result = cv2.bitwise_and(nemo
>>> plt.subplot(1, 2, 1)
>>> plt.imshow(final_mask, cmap="gray")
>>> plt.subplot(1, 2, 2)
>>> plt.imshow(final_result)
>>> plt.show()
```

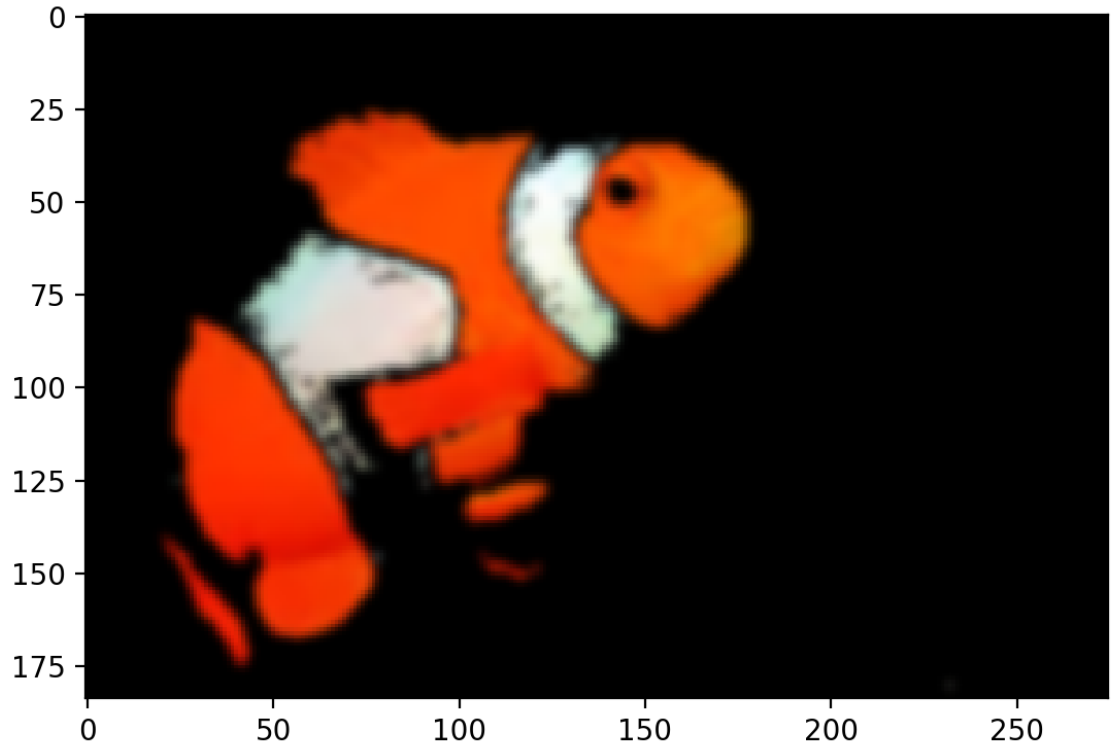



Essentially, you have a rough segmentation of Nemo in HSV color space. You'll notice there are a few stray pixels along the segmentation border, and if you like, you can use a Gaussian blur to tidy up the small false detections.

A Gaussian blur is an image filter that uses a kind of function called a Gaussian to transform each pixel in the image. It has the result of smoothing out image noise and reducing detail. Here's what applying the blur looks like for our image:

Python >>>

```
>>> blur = cv2.GaussianBlur(final_result, (7, 7), 0)
>>> plt.imshow(blur)
>>> plt.show()
```



Does This Segmentation Generalize to Nemo's Relatives?

Just for fun, let's see how well this segmentation technique generalizes to other clownfish images. In the repository, there's a selection of six images of clownfish from Google, licensed for public use. The images are in a subdirectory and indexed nemo*i*.jpg, where *i* is the index from `0-5`.

First, load all Nemo's relatives into a list:

Python

```
path = "./images/nemo"

nemos_friends = []
for i in range(6):
    friend = cv2.cvtColor(cv2.imread(pat
    nemos_friends.append(friend)
```


You can combine all the code used above to return the segmented image. Expand this se

The Segment Fish Function

With that useful function, you can then segr

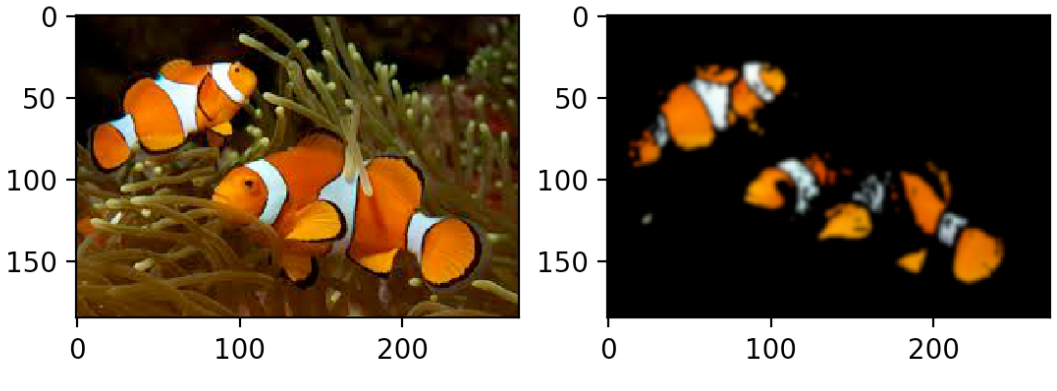
Python

```
results = [segment_fish(friend) for friend in nemos_friends]
```

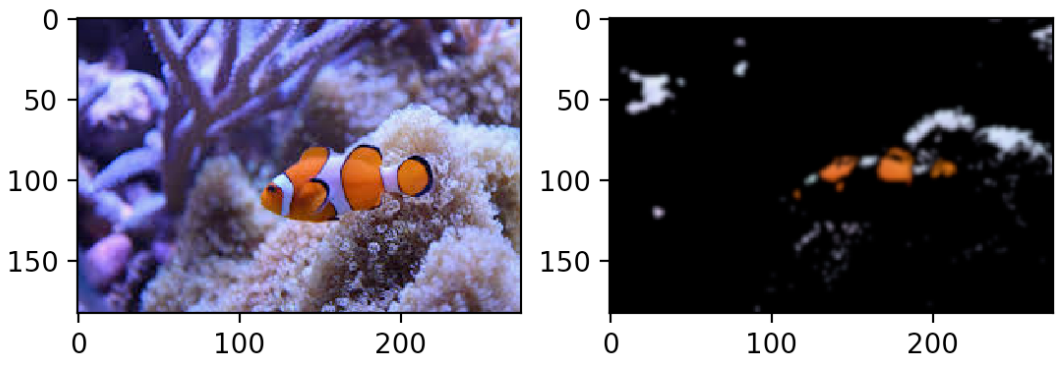
Let's view all the results by plotting them in a loop:

Python

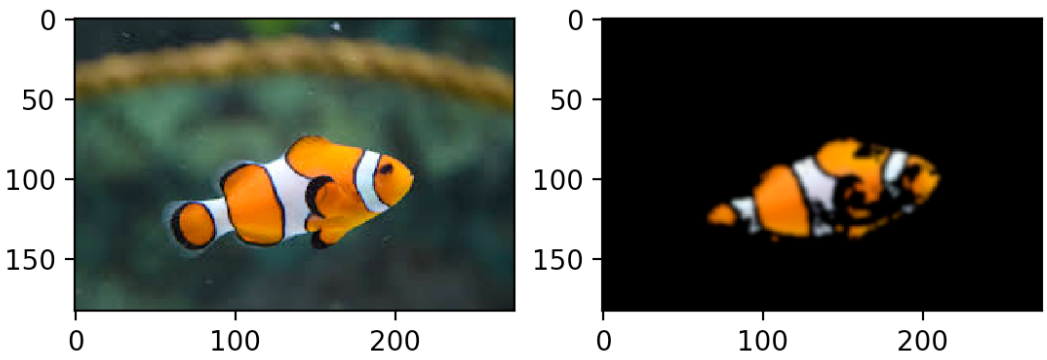
```
for i in range(1, 6):
    plt.subplot(1, 2, 1)
    plt.imshow(nemos_friends[i])
    plt.subplot(1, 2, 2)
    plt.imshow(results[i])
    plt.show()
```

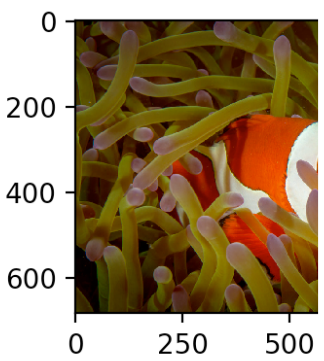


The foreground clownfish has orange shades darker than our range.



The shadowed bottom half of Nemo's nephew is completely excluded, but bits of the purple anemone in the background look awfully like Nemo's blue tinged stripes…



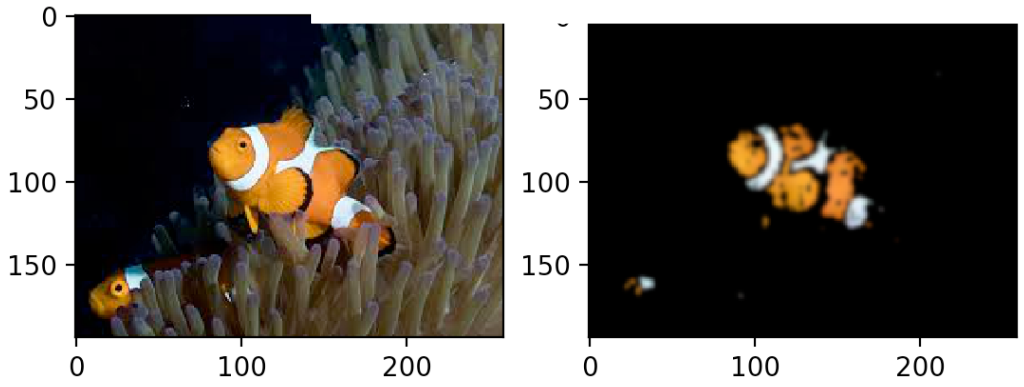



```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Overall, this simple segmentation method has successfully located the majority of Nemo's relatives. It is clear, however, that segmenting one clownfish with particular lighting and background may not necessarily generalize well to segmenting all clownfish.

Conclusion

In this tutorial, you've seen what a few different color spaces are, how an image is distributed across RGB and HSV color spaces, and how to use OpenCV to convert between color spaces and segment out ranges.

Altogether, you've learned how a basic understanding of how color spaces in OpenCV can be used to perform object segmentation in images, and hopefully seen its potential for doing other tasks as well. Where lighting and background are controlled, such as in an experimental setting or with a more homogeneous dataset, this segmentation technique is simple, fast, and reliable.

About **Rebecca Stone**

Rebecca is a PhD student in computer vision and artificial intelligence applied to medical images. She's passionate about teaching.

» More about Rebecca

Each tutorial at Real Python is created by worked on this tutorial are:

Adriana

What Do You Think?

Tweet Share Email

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won't make the cut here.

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Keep Learning

Related Tutorial Categories: intermediate machine-learning

All Tutorial Topics

advanced api basics best-practices community databases data-science devops django docker flask front-end intermediate machine-learning python testing tools web-dev web-scraping

Table of Contents

Conclusion