Russ Islam

19 Followers About Follow

Implementing SIFT in Python: A Complete Guide (Part 1)

Dive into the details and solidify your computer vision fundamentals

Russ Islam Feb 17, 2020 · 10 min read

It's a classic in computer vision. Many computer vision engineers rely on it everyday, but most of them have only a high-level understanding of it, or worse, they've only seen a couple of slides on the topic back in college. You don't want to be that engineer.

I'm talking about the scale-invariant feature transform (SIFT). We're going to swim through the mud and pick apart all the gory details. It's in the gory details where all the learning happens — and I'm going to walk you through it step by step. This is the first part of a two-part tutorial (find Part 2 here).

Remember, no one becomes a good engineer or scientist by dealing with things only at a high level.

Deep learning is great and all, and there's no denying that it's the state of the art on several tasks, but remember that it's not a panacea. You can get good results using deep learning for many applications, but there are plenty of other applications out there (***cough** robotics*) where algorithms based on advanced math and physics are just plain better. You've got to be agnostic when it comes to finding the right solution to the problem at hand. If all you've got is a hammer, every problem looks like a nail, and if all you've got is a deep learning hammer, every problem looks like a deep learning nail. But what happens if (and when) you come across problems you can't solve with just deep learning?

Then it's time to bust out the big guns — classic algorithms like SIFT.

But it's about more than just solving problems. It's about developing your skills. Even if you can get better results with deep learning, you don't want to skip out on learning things like SIFT. Learning the details behind SIFT will teach you profound, fundamental truths that underlie computer vision. Think of it this way: even if you can write programs with deep learning algorithms that can solve a handful of mechanics problems more accurately than Newtonian mechanics can, that doesn't mean you shouldn't learn Newtonian mechanics. Would you rather be a physicist who can skillfully use a variety of algorithms to tackle any problem, or a 9-to-5 programmer who only knows how to call some neural network APIs?

It's time to dive into the mud.

The Code

You can find my Python implementation of SIFT here. In this tutorial, we'll walk through this code (the file `pysift.py`) step by step, printing and visualizing variables along the way to help us fully understand what's happening at every moment. I wrote this implementation by closely following the OpenCV implementation, simplifying and Pythonizing the logic without sacrificing any details.

The usage is simple:

```python
import cv2
import pysift

image = cv2.imread('your_image.png', 0)
keypoints, descriptors = pysift.computeKeypointsAndDescriptors(image)
```

pysift_usage.py hosted with ❤ by GitHub view raw

Simply pass a 2D NumPy array to `computeKeypointsAndDescriptors()` to return a list of OpenCV `KeyPoint` objects and a list of the associated 128-length descriptor vectors. This way `pysift` works as a drop-in replacement for OpenCV's SIFT functionality. Note that this implementation is meant to be clear and easy to understand, and it's not designed for high performance. It'll take a couple of minutes to process an ordinary input image on an ordinary laptop.

Clone the repo and try out the template matching demo. You'll get almost the same keypoints you'd get using OpenCV (the differences are due to floating point error).

SIFT Theory and Overview

Let's briefly go over the reasoning behind SIFT and develop a high-level roadmap of the algorithm. I won't dwell on the math. You can (and should) read the original paper here.

SIFT identifies keypoints that are distinctive across an image's width, height, and most importantly, scale. By considering scale, we can identify keypoints that will remain stable (to an extent) even when the template of interest changes size, when the image quality becomes better or worse, or when the template undergoes changes in viewpoint or aspect ratio. Moreover, each keypoint has an associated orientation that makes SIFT features invariant to template rotations. Finally, SIFT will generate a descriptor for each keypoint, a 128-length vector that allows keypoints to be compared. These descriptors are nothing more than a histogram of gradients computed within the keypoint's neighborhood.

Most of the tricky details in SIFT relate to scale space, like applying the correct amount of blur to the input image, or converting keypoints from one scale to another.

Below you can see `pysift`'s main function, `computeKeypointsAndDescriptors()`, which gives you a clear overview of the different components involved in SIFT. First, we call `generateBaseImage()` to appropriately blur and double the input image to produce the base image of our "image pyramid", a set of successively blurred and downsampled images that form our scale space. We then call `computeNumberOfOctaves()` to compute the number of layers ("octaves") in our image pyramid. Now we can actually build the image pyramid. We start with `generateGaussianKernels()` to create a list of scales (gaussian kernel sizes) that is passed to `generateGaussianImages()`, which repeatedly blurs and downsamples the base image. Next we subtract adjacent pairs of gaussian images to form a pyramid of difference-of-Gaussian ("DoG") images. We'll use this final DoG image pyramid to identify keypoints using `findScaleSpaceExtrema()`. We'll clean up these keypoints by removing duplicates and converting them to the input image size. Finally, we'll generate descriptors for each keypoint via `generateDescriptors()`.

```
from numpy import all, any, array, arctan2, cos, sin, exp, dot, log, logical_and, roll,
from numpy.linalg import det, lstsq, norm
```

```python
from cv2 import resize, GaussianBlur, subtract, KeyPoint, INTER_LINEAR, INTER_NEAREST
from functools import cmp_to_key
import logging

####################
# Global variables #
####################

logger = logging.getLogger(__name__)
float_tolerance = 1e-7

#################
# Main function #
#################

def computeKeypointsAndDescriptors(image, sigma=1.6, num_intervals=3, assumed_blur=0.5,
    """Compute SIFT keypoints and descriptors for an input image
    """
    image = image.astype('float32')
    base_image = generateBaseImage(image, sigma, assumed_blur)
    num_octaves = computeNumberOfOctaves(base_image.shape)
    gaussian_kernels = generateGaussianKernels(sigma, num_intervals)
    gaussian_images = generateGaussianImages(base_image, num_octaves, gaussian_kernels
    dog_images = generateDoGImages(gaussian_images)
    keypoints = findScaleSpaceExtrema(gaussian_images, dog_images, num_intervals, sigma
    keypoints = removeDuplicateKeypoints(keypoints)
    keypoints = convertKeypointsToInputImageSize(keypoints)
    descriptors = generateDescriptors(keypoints, gaussian_images)
    return keypoints, descriptors
```

main_function.py hosted with ❤ by GitHub view raw

Simple enough. Now we'll explore these functions one at a time.

Scale Space and Image Pyramids

```python
def generateBaseImage(image, sigma, assumed_blur):
    """Generate base image from input image by upsampling by 2 in both directions and b
    """
    logger.debug('Generating base image...')
    image = resize(image, (0, 0), fx=2, fy=2, interpolation=INTER_LINEAR)
    sigma_diff = sqrt(max((sigma ** 2) - ((2 * assumed_blur) ** 2), 0.01))
    return GaussianBlur(image, (0, 0), sigmaX=sigma_diff, sigmaY=sigma_diff)  # the ima

def computeNumberOfOctaves(image_shape):
    """Compute number of octaves in image pyramid as function of base image shape (Oper
    """
    return int(round(log(min(image_shape)) / log(2) - 1))

def generateGaussianKernels(sigma, num_intervals):
    """Generate list of gaussian kernels at which to blur the input image. Default valu
    """
    logger.debug('Generating scales...')
    num_images_per_octave = num_intervals + 3
    k = 2 ** (1. / num_intervals)
    gaussian_kernels = zeros(num_images_per_octave)  # scale of gaussian blur necessary
    gaussian_kernels[0] = sigma

    for image_index in range(1, num_images_per_octave):
        sigma_previous = (k ** (image_index - 1)) * sigma
        sigma_total = k * sigma_previous
        gaussian_kernels[image_index] = sqrt(sigma_total ** 2 - sigma_previous ** 2)
    return gaussian_kernels

def generateGaussianImages(image, num_octaves, gaussian_kernels):
    """Generate scale-space pyramid of Gaussian images
    """
    logger.debug('Generating Gaussian images...')
    gaussian_images = []

    for octave_index in range(num_octaves):
        gaussian_images_in_octave = []
        gaussian_images_in_octave.append(image)  # first image in octave already has th
        for gaussian_kernel in gaussian_kernels[1:]:
            image = GaussianBlur(image, (0, 0), sigmaX=gaussian_kernel, sigmaY=gaussian
            gaussian_images_in_octave.append(image)
        gaussian_images.append(gaussian_images_in_octave)
        octave_base = gaussian_images_in_octave[-3]
        image = resize(octave_base, (int(octave_base.shape[1] / 2), int(octave_base.sha
    return array(gaussian_images)

def generateDoGImages(gaussian_images):
    """Generate Difference-of-Gaussians image pyramid
    """
    logger.debug('Generating Difference-of-Gaussian images...')
    dog_images = []

    for gaussian_images_in_octave in gaussian_images:
        dog_images_in_octave = []
        for first_image, second_image in zip(gaussian_images_in_octave, gaussian_images
            dog_images_in_octave.append(subtract(second_image, first_image))  # ordinar
        dog_images.append(dog_images_in_octave)
    return array(dog_images)
```

Our first step is `generateBaseImage()`, which simply doubles the input image in size and applies Gaussian blur. Assuming the input image has a blur of `assumed_blur = 0.5`, if we want our resulting base image to have a blur of `sigma`, we need to blur the doubled input image by `sigma_diff`. Note that blurring an input image by kernel size σ_1 and then blurring the resulting image by σ_2 is equivalent to blurring the input image just once by σ, where $\sigma^2 = \sigma_1^2 + \sigma_2^2$. (Interested readers can find a proof here.)

Now let's look at `computeNumberOfOctaves()`, which is simple but requires some explanation. This function computes the number of times we can repeatedly halve an image until it becomes too small. Well, for starters, the final image should have a side length of at least 1 pixel. We can set up an equation for this. If y is the shorter side length of the image, then we have $y / 2^x = 1$, where x is the number of times we can halve the base image. We can take the logarithm of both sides and solve for x to obtain $\log(y) / \log(2)$. So why does the -1 show up in the function above? At the end of the day, we have to round x down to the nearest integer (floor(x)) to have an integer number of layers in our image pyramid.

Actually, if you look at how `numOctaves` is used in the functions below, we halve the base image `numOctaves – 1` times to end up with `numOctaves` layers, including the base image. This ensures the image in the highest octave (the smallest image) will have a side length of at least 3. This is important because we'll search for minima and maxima in each DoG image later, which means we need to consider 3-by-3 pixel neighborhoods.

Next we `generateGaussianKernels()`, which creates a list of the amount of blur for each image in a particular layer. Note that the image pyramid has `numOctaves` layers, but each layer itself has `numIntervals + 3` images. All the images in the same layer have the same width and height, but the amount of blur successively increases. Where does the `+ 3` come from? We have `numIntervals + 1` images to cover `numIntervals` steps from one blur value to twice that value. We have another `+ 2` for one blur step before the first image in the layer and another blur step after the last image in the layer. We need these two extra images at the end because we'll subtract adjacent Gaussian images to create a DoG image pyramid. This means that if we compare images from two neighboring layers, we'll see many of the same blur values repeated. We need this repetition to make sure we cover all blur steps when we subtract images.

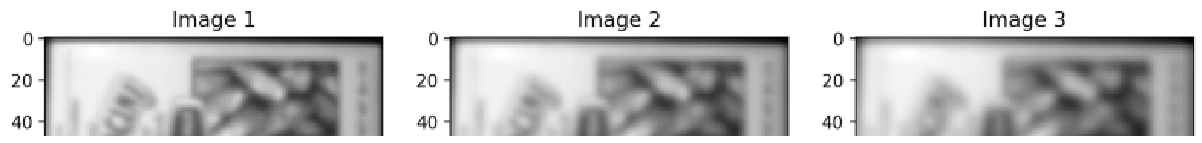
Let's stop for a minute and print out the generated kernels:

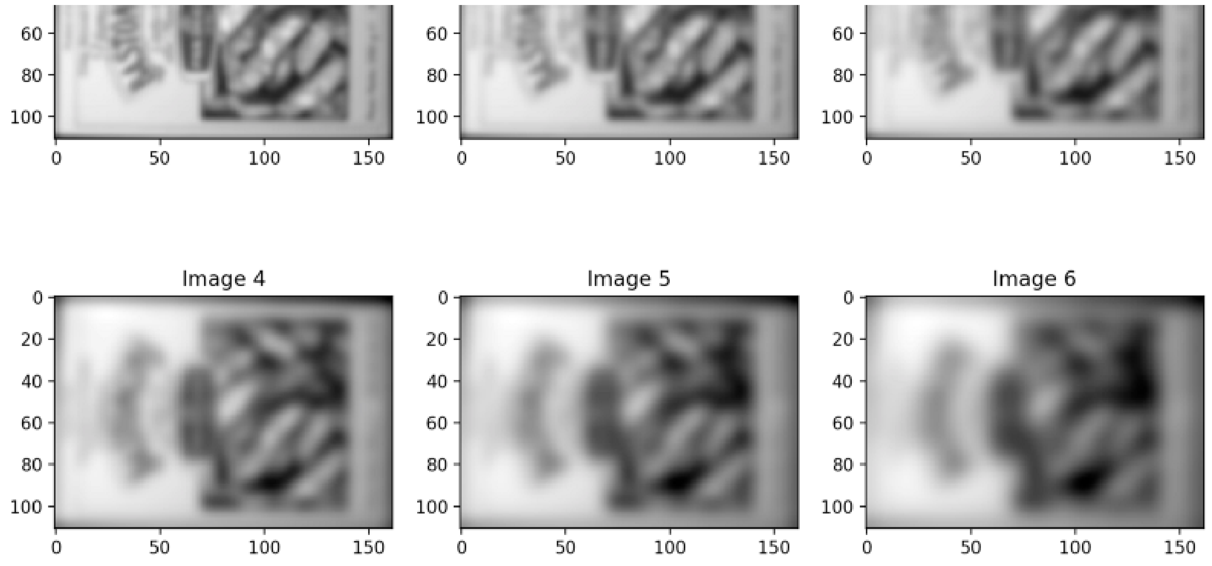
```
print(gaussian_kernels)
array([1.6, 1.22627, 1.54501, 1.94659, 2.45255, 3.09002])
```

That's weird — how come we drop from `1.6` to `1.22627` before increasing again? Take another good look at the `generateGaussianKernels()` function. The first element of this array is simply our starting `sigma`, but after that each element is the additional scale we need to convolve with the previous scale. To be concrete, we start out with an image with scale `1.6`. We blur this image with a Gaussian kernel of `1.22627`, which produces an image with a blur of `sqrt(1.6 ** 2 + 1.22627 ** 2) == 2.01587` , and we blur this new image with a kernel of size `1.54501` to produce a third image of blur `sqrt(2.01587 ** 2 + 1.54501 ** 2) == 2.53984`. Finally, we blur this image by `1.94659` to produce our last image, which has a blur of `sqrt(2.53984 ** 2 + 1.94659 ** 2) == 3.2`. But `2 * 1.6 == 3.2`, so we've moved up exactly one octave!

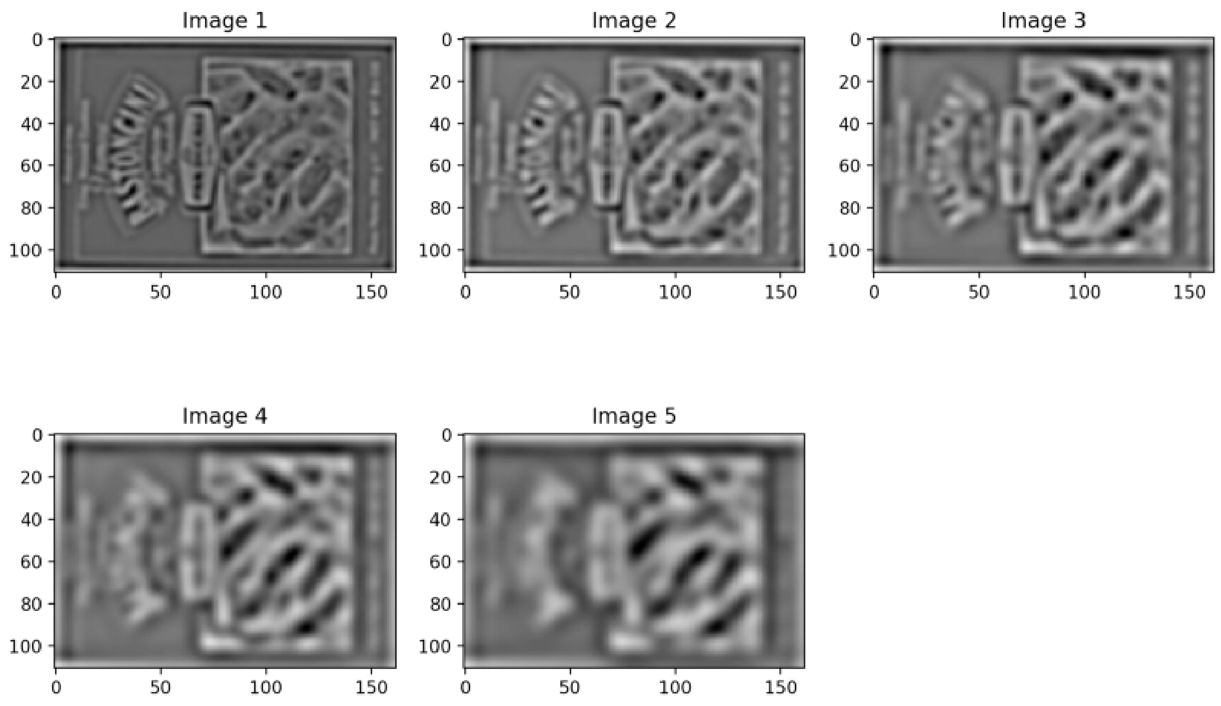
Now we have all we need to actually generate our image pyramids. We `generateGaussianImages()` by starting with our base image and successively blurring it according to our `gaussian_kernels`. Note that we skip the first element of `gaussian_kernels` because we begin with an image that already has that blur value. We halve the third-to-last image, since this has the appropriate blur we want, and use this to begin the next layer. This way we get that nice overlap mentioned previously. Finally, we `generateDoGImages()` by subtracting successive pairs of these Gaussian-blurred images. Careful — although ordinary subtraction will work here because we've cast the input image to `float32`, we'll use OpenCV's `subtract()` function so that the code won't break if you choose to remove this cast and pass in `uint` type images.

Below I've plotted the third layer of our Gaussian pyramid, `gaussian_images[2]`. Note how the images get progressively smoother, while finer features disappear. I've also plotted `dog_images[2]`. Remember that `dog_images[2][i] = gaussian_images[2][i + 1] – gaussian_images[2][i]`, and notice how the DoG images look like edge maps.





Images from the third layer of our Gaussian image pyramid.



Images from the third layer of our difference-of-Gaussians image pyramid.

Run the code and plot these images yourself to verify that you obtain similar results.

At last we've got our DoG image pyramid. It's time to find our keypoints.

Finding Scale Space Extrema

```
def findScaleSpaceExtrema(gaussian_images, dog_images, num_intervals, sigma, image_bor
    """Find pixel positions of all scale-space extrema in the image pyramid
    """
    logger.debug('Finding scale-space extrema...')
    threshold = floor(0.5 * contrast_threshold / num_intervals * 255)  # from OpenCV ir
    keypoints = []

    for octave_index, dog_images_in_octave in enumerate(dog_images):
        for image_index, (first_image, second_image, third_image) in enumerate(zip(dog_
            # (i, j) is the center of the 3x3 array
            for i in range(image_border_width, first_image.shape[0] - image_border_widt
                for j in range(image_border_width, first_image.shape[1] - image_border_
                    if isPixelAnExtremum(first_image[i-1:i+2, j-1:j+2], second_image[i-
                        localization_result = localizeExtremumViaQuadraticFit(i, j, ima
                        if localization_result is not None:
                            keypoint, localized_image_index = localization_result
                            keypoints_with_orientations = computeKeypointsWithOrientati
                            for keypoint_with_orientation in keypoints_with_orientatior
                                keypoints.append(keypoint_with_orientation)
    return keypoints

def isPixelAnExtremum(first_subimage, second_subimage, third_subimage, threshold):
    """Return True if the center element of the 3x3x3 input array is strictly greater 1
    """
    center_pixel_value = second_subimage[1, 1]
    if abs(center_pixel_value) > threshold:
        if center_pixel_value > 0:
            return all(center_pixel_value >= first_subimage) and \
                   all(center_pixel_value >= third_subimage) and \
                   all(center_pixel_value >= second_subimage[0, :]) and \
                   all(center_pixel_value >= second_subimage[2, :]) and \
                   center_pixel_value >= second_subimage[1, 0] and \
                   center_pixel_value >= second_subimage[1, 2]
        elif center_pixel_value < 0:
            return all(center_pixel_value <= first_subimage) and \
                   all(center_pixel_value <= third_subimage) and \
                   all(center_pixel_value <= second_subimage[0, :]) and \
                   all(center_pixel_value <= second_subimage[2, :]) and \
                   center_pixel_value <= second_subimage[1, 0] and \
                   center_pixel_value <= second_subimage[1, 2]
    return False
```

scale_space_extrema_functions.py hosted with ❤ by GitHub view raw

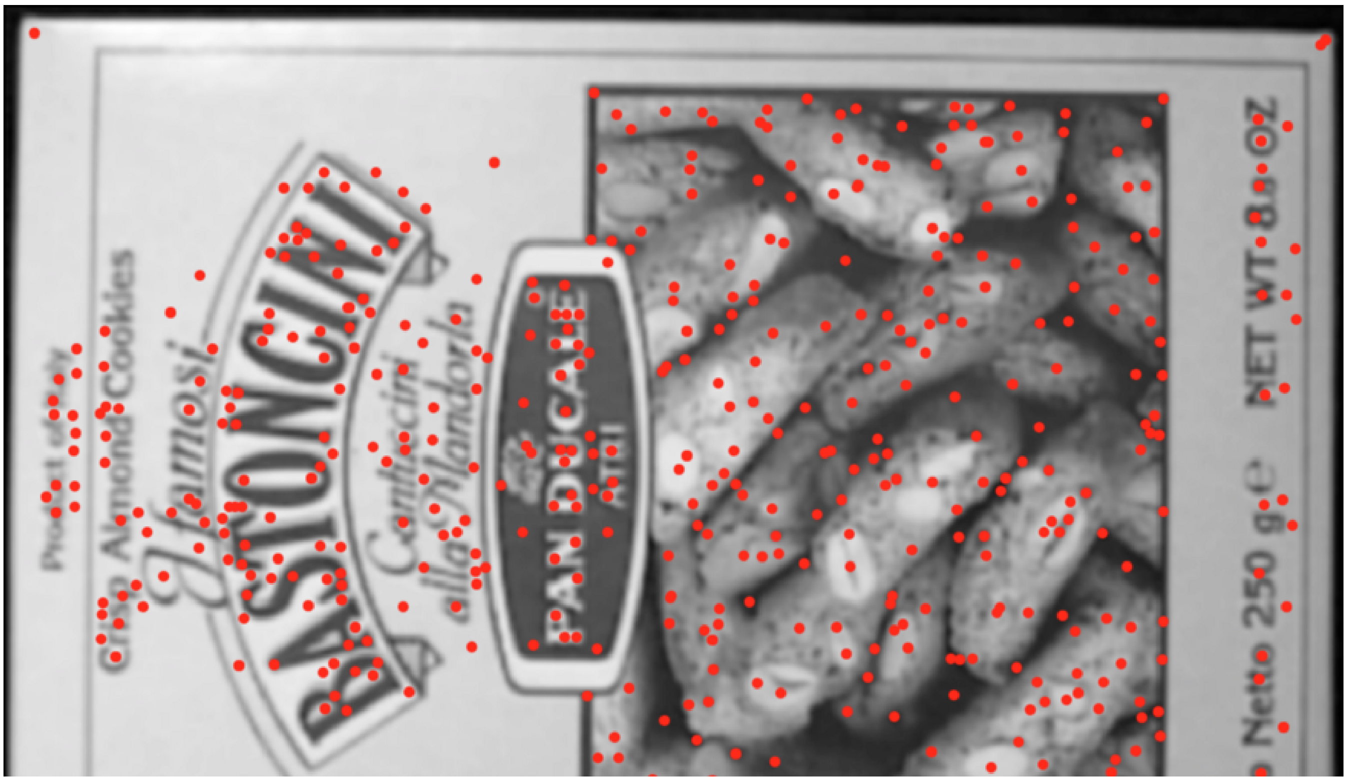
This part's easy. We just iterate through each layer, taking three successive images at a time. Remember that all images in a layer have the same size — only their amounts of blur differ. In each triplet of images, we look for pixels in the middle image that are greater than or less than all of their 26 neighbors: 8 neighbors in the middle image, 9 neighbors in the image below, and 9 neighbors in the image above. The function `isPixelAnExtremum()` performs this check. These are our maxima and minima (strictly speaking, they include saddle points because we include pixels that are equal in value to all their neighbors). When we've found an extremum, we localize its position at the subpixel level along all three dimensions (width, height, and scale) using `localizeExtremumViaQuadraticFit()`, explained in more detail below.

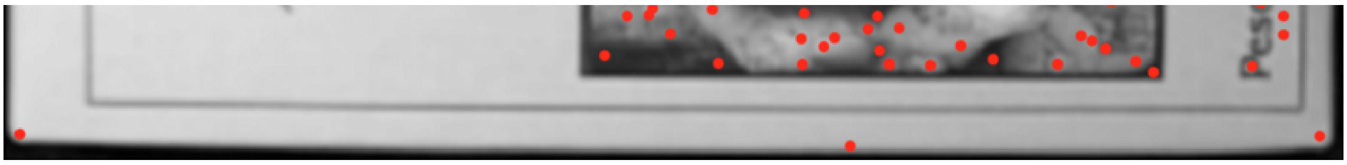
Our last step here is to compute orientations for each keypoint, which we'll cover in Part 2 of this tutorial. There may be more than one orientation, so we create and append a new keypoint for each one.

After localization, we check that the keypoint's new pixel location has enough contrast. I'll spare you the math because the SIFT paper explains this in good detail. In a nutshell, the ratio of eigenvalues of the 2D Hessian along the width and height of the keypoint's image gives us contrast information.

One last note: the keypoint's position (`keypoint.pt`) is repeatedly doubled according to its layer so that it corresponds to coordinates in the base image. The `keypoint.octave` and `keypoint.size` (i.e., scale) attributes are defined according to the OpenCV implementation. These two will come in handy later.

You can try plotting your keypoints over the base image to visualize the pixels SIFT finds interesting. I've done so below on `box.png` from the repo:




Our initial keypoints plotted over the base image.

Localizing Extrema

```
def localizeExtremumViaQuadraticFit(i, j, image_index, octave_index, num_intervals, dog
    """Iteratively refine pixel positions of scale-space extrema via quadratic fit arou
    """
    logger.debug('Localizing scale-space extrema...')
    extremum_is_outside_image = False
    image_shape = dog_images_in_octave[0].shape
    for attempt_index in range(num_attempts_until_convergence):
        # need to convert from uint8 to float32 to compute derivatives and need to resc
        first_image, second_image, third_image = dog_images_in_octave[image_index-1:ima
        pixel_cube = stack([first_image[i-1:i+2, j-1:j+2],
                            second_image[i-1:i+2, j-1:j+2],
                            third_image[i-1:i+2, j-1:j+2]]).astype('float32') / 255.
        gradient = computeGradientAtCenterPixel(pixel_cube)
        hessian = computeHessianAtCenterPixel(pixel_cube)
        extremum_update = -lstsq(hessian, gradient, rcond=None)[0]
        if abs(extremum_update[0]) < 0.5 and abs(extremum_update[1]) < 0.5 and abs(extr
            break
        j += int(round(extremum_update[0]))
        i += int(round(extremum_update[1]))
        image_index += int(round(extremum_update[2]))
        # make sure the new pixel_cube will lie entirely within the image
        if i < image_border_width or i >= image_shape[0] - image_border_width or j < ir
            extremum_is_outside_image = True
            break
    if extremum_is_outside_image:
        logger.debug('Updated extremum moved outside of image before reaching converger
        return None
    if attempt_index >= num_attempts_until_convergence - 1:
        logger.debug('Exceeded maximum number of attempts without reaching convergence
        return None
    functionValueAtUpdatedExtremum = pixel_cube[1, 1, 1] + 0.5 * dot(gradient, extremur
    if abs(functionValueAtUpdatedExtremum) * num_intervals >= contrast_threshold:
        xy_hessian = hessian[:2, :2]
        xy_hessian_trace = trace(xy_hessian)
        xy_hessian_det = det(xy_hessian)
        if xy_hessian_det > 0 and eigenvalue_ratio * (xy_hessian_trace ** 2) < ((eigenv
            # Contrast check passed -- construct and return OpenCV KeyPoint object
            keypoint = KeyPoint()
            keypoint.pt = ((j + extremum_update[0]) * (2 ** octave_index), (i + extremu
            keypoint.octave = octave_index + image_index * (2 ** 8) + int(round((extre
            keypoint.size = sigma * (2 ** ((image_index + extremum_update[2]) / float3
            keypoint.response = abs(functionValueAtUpdatedExtremum)
            return keypoint, image_index
    return None

def computeGradientAtCenterPixel(pixel_array):
    """Approximate gradient at center pixel [1, 1, 1] of 3x3x3 array using central dif
    """
    # With step size h, the central difference formula of order O(h^2) for f'(x) is (f
    # Here h = 1, so the formula simplifies to f'(x) = (f(x + 1) - f(x - 1)) / 2
    # NOTE: x corresponds to second array axis, y corresponds to first array axis, and
    dx = 0.5 * (pixel_array[1, 1, 2] - pixel_array[1, 1, 0])
    dy = 0.5 * (pixel_array[1, 2, 1] - pixel_array[1, 0, 1])
    ds = 0.5 * (pixel_array[2, 1, 1] - pixel_array[0, 1, 1])
    return array([dx, dy, ds])

def computeHessianAtCenterPixel(pixel_array):
    """Approximate Hessian at center pixel [1, 1, 1] of 3x3x3 array using central diffe
    """
    # With step size h, the central difference formula of order O(h^2) for f''(x) is (
    # Here h = 1, so the formula simplifies to f''(x) = f(x + 1) - 2 * f(x) + f(x - 1)
    # With step size h, the central difference formula of order O(h^2) for (d^2) f(x,
    # Here h = 1, so the formula simplifies to (d^2) f(x, y) / (dx dy) = (f(x + 1, y +
    # NOTE: x corresponds to second array axis, y corresponds to first array axis, and
    center_pixel_value = pixel_array[1, 1, 1]
    dxx = pixel_array[1, 1, 2] - 2 * center_pixel_value + pixel_array[1, 1, 0]
    dyy = pixel_array[1, 2, 1] - 2 * center_pixel_value + pixel_array[1, 0, 1]
    dss = pixel_array[2, 1, 1] - 2 * center_pixel_value + pixel_array[0, 1, 1]
    dxy = 0.25 * (pixel_array[1, 2, 2] - pixel_array[1, 2, 0] - pixel_array[1, 0, 2] +
    dxs = 0.25 * (pixel_array[2, 1, 2] - pixel_array[2, 1, 0] - pixel_array[0, 1, 2] +
    dys = 0.25 * (pixel_array[2, 2, 1] - pixel_array[2, 0, 1] - pixel_array[0, 2, 1] +
    return array([[dxx, dxy, dxs],
                  [dxy, dyy, dys],
                  [dxs, dys, dss]])
```

The code to localize a keypoint may look involved, but it's actually pretty straightforward. It implements verbatim the localization procedure described in the original SIFT paper. We fit a quadratic model to the input keypoint pixel and all 26 of its neighboring pixels (we call this a `pixel_cube`). We update the keypoint's position with the subpixel-accurate extremum estimated from this model. We iterate at most 5 times until the next update moves the keypoint less than 0.5 in any of the three directions. This means the quadratic model has converged to one pixel location. The two helper functions `computeGradientAtCenterPixel()` and `computeHessianAtCenterPixel()` implement second-order central finite difference approximations of the gradients and hessians in all three dimensions. If you need a refresher on finite differences (or if you're learning them for the first time!), take a look here or here. The key takeaway is that ordinary quadratic interpolation that you may have done in calculus class won't work well here because we're using a uniform mesh (a grid of evenly spaced pixels). Finite difference approximations take into account this discretization to produce more accurate extrema estimates.

Now we've found our keypoints and have accurately localized them. We've gotten far, but there are two big tasks left: computing orientations and generating descriptors.

We'll handle these in Part 2. See you there.

Python Computer Vision Sift Image Processing Template Matching