Can you do it without looking?

Fully Homomorphic Image Processing for SIFT

TEAM : COMPARE AND CONTRAST

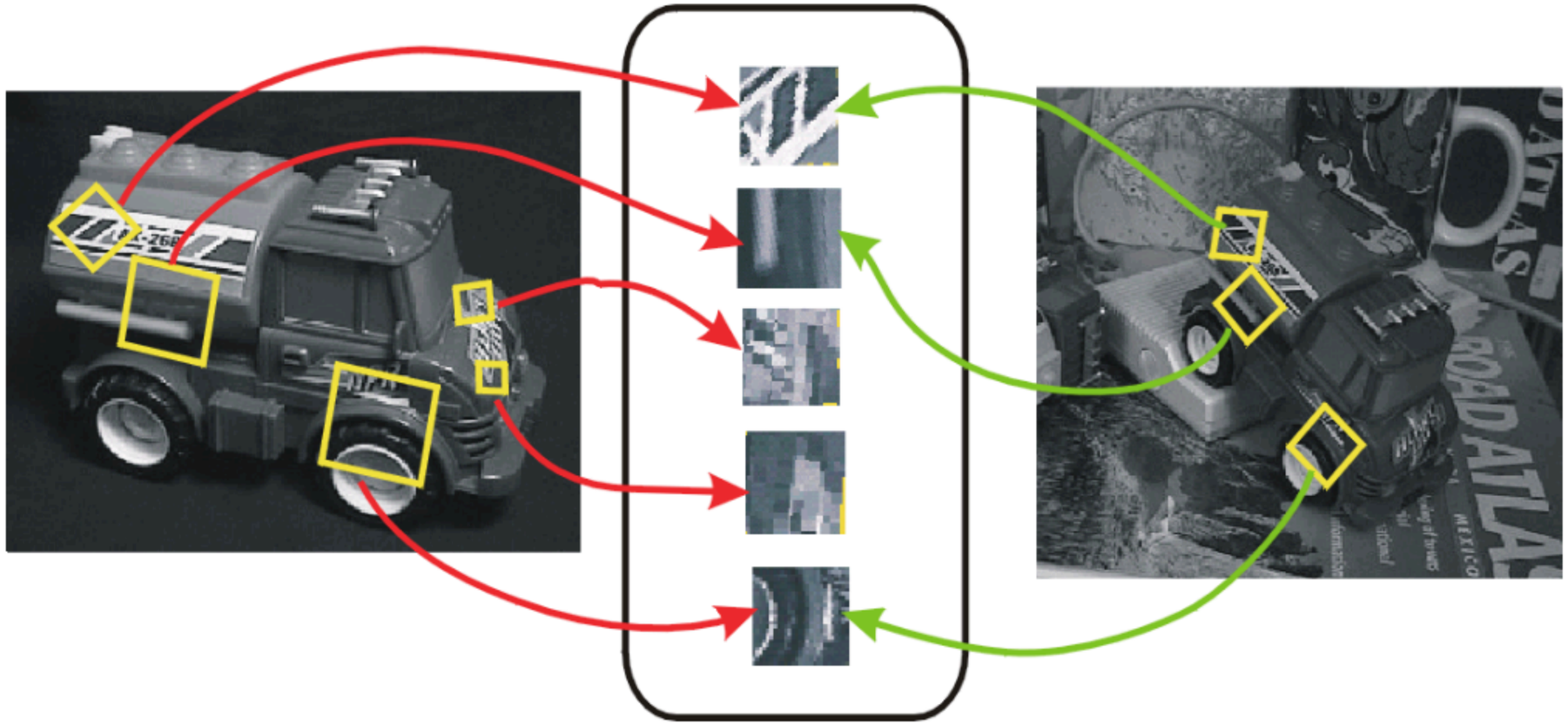
Introducing SIFT

Scale Invariant Feature Transform

SIFT

SIFT: Scale Invariant Feature Transform

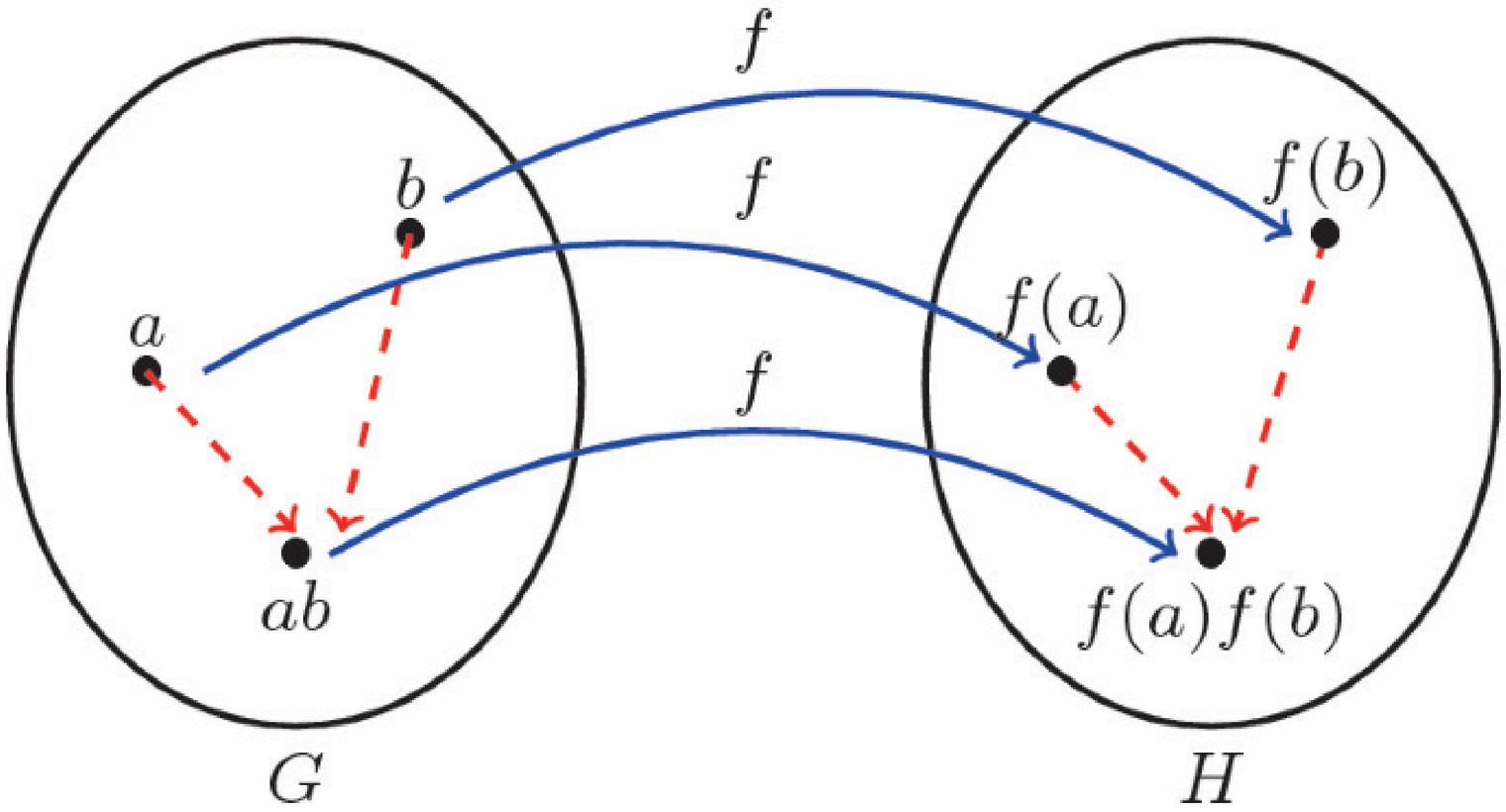
Recover features with change of **position, orientation** and **scale**



Introducing FHE

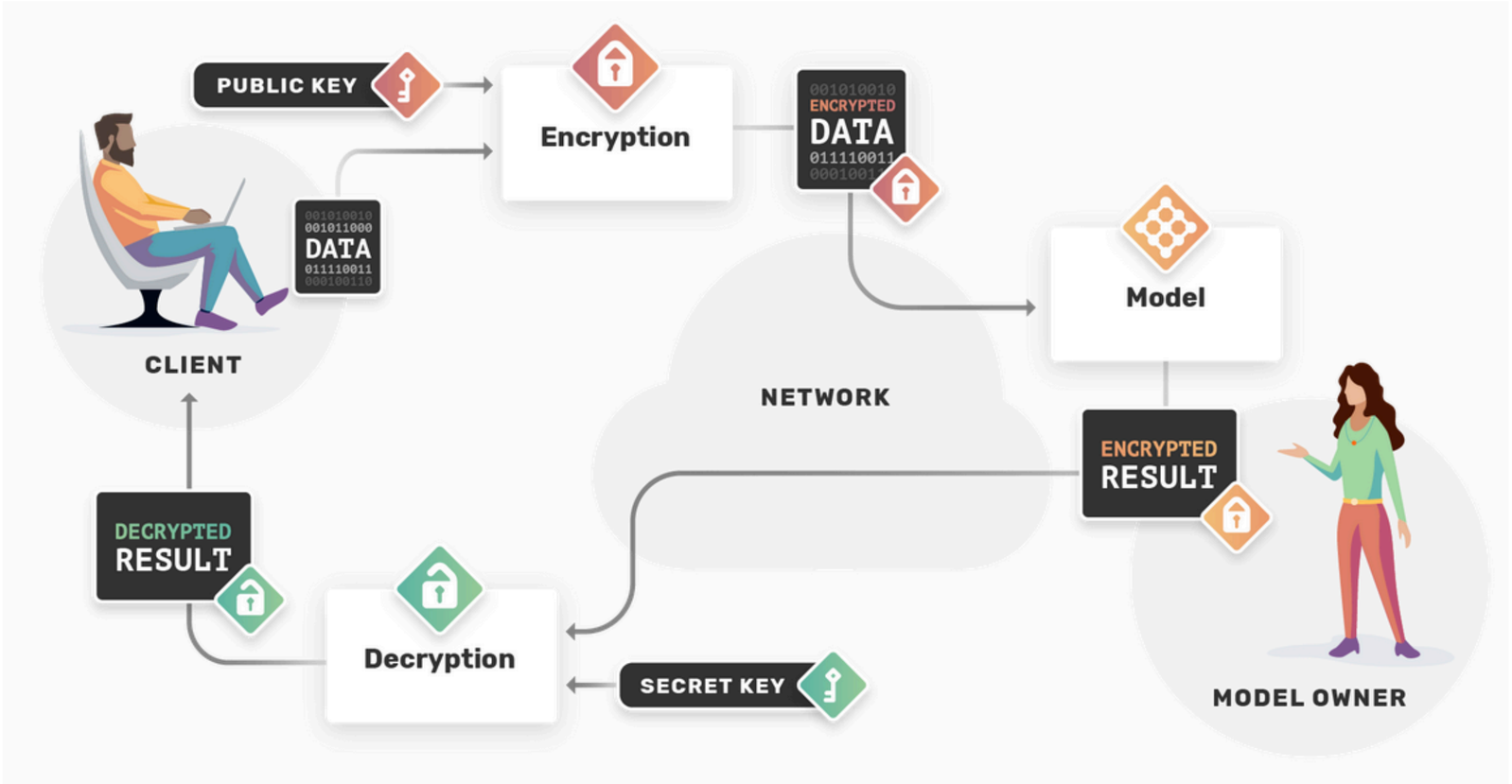
Fully Homomorphic Encryption

Homomorphic Operations



FHE Abstractions

HE ensures that performing operations on encrypted data and decrypting the result is equivalent to performing analogous operations without any encryption



Primitives Available

- Addition of floats
- Multiplication of floats
- Array of floats
- Indexing if we the indexes are unencrypted.

Problem Statement

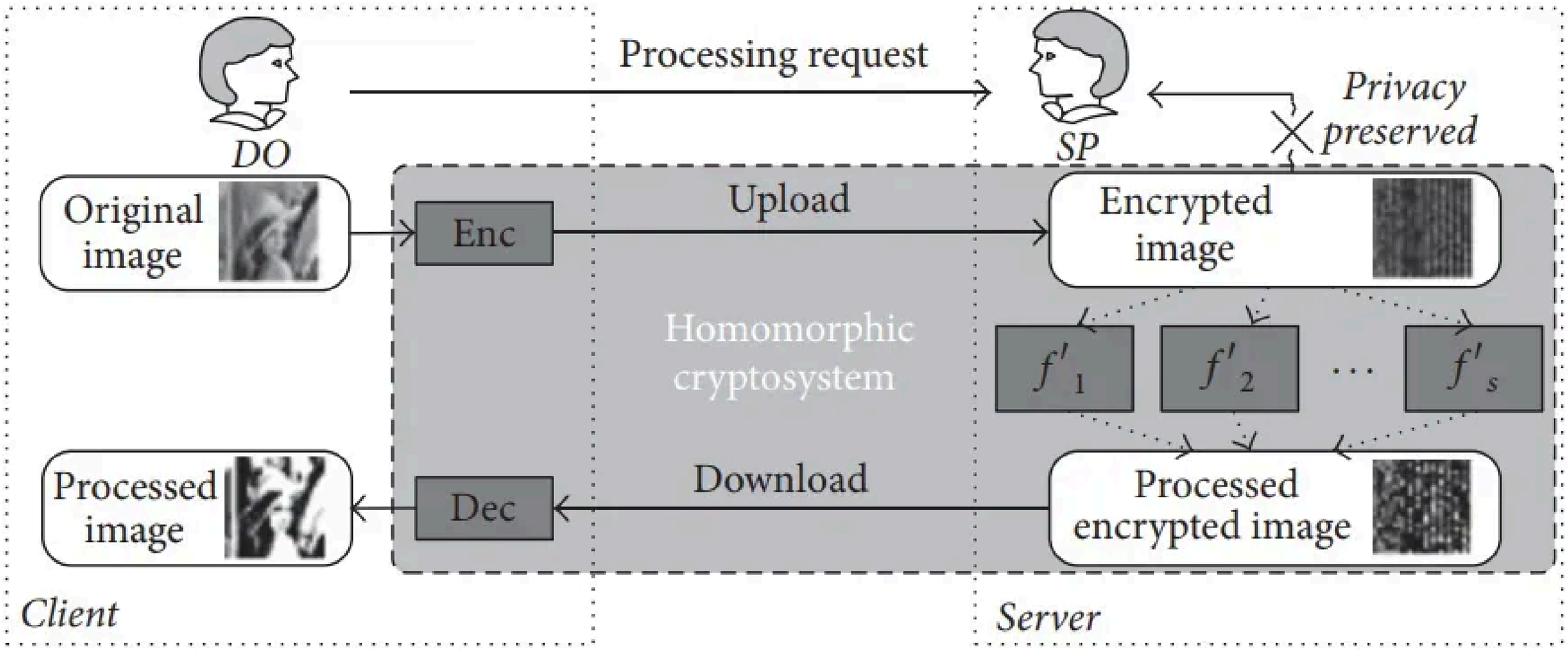
The problem is to perform SIFT on an encrypted image and produce keypoints which on decryption will give keypoints with their location, orientation and their descriptors

Note : Even the number of keypoints needs to be hidden before decryption

Motivation

You want to do expensive/heavy image processing on your images, but your device is not powerful enough. You can send it to a cloud service to do the processing, but you don't want to send personal information. What do you do? We propose using homomorphic encryption, which lets us perform computation on data that is encrypted.

We picked SIFT as a way to demonstrate the feasibility of modern fully homomorphic encryption (FHE) for image processing. This is because the various concepts involved in the SIFT pose interesting and sometimes new unseen challenges of working with FHE.

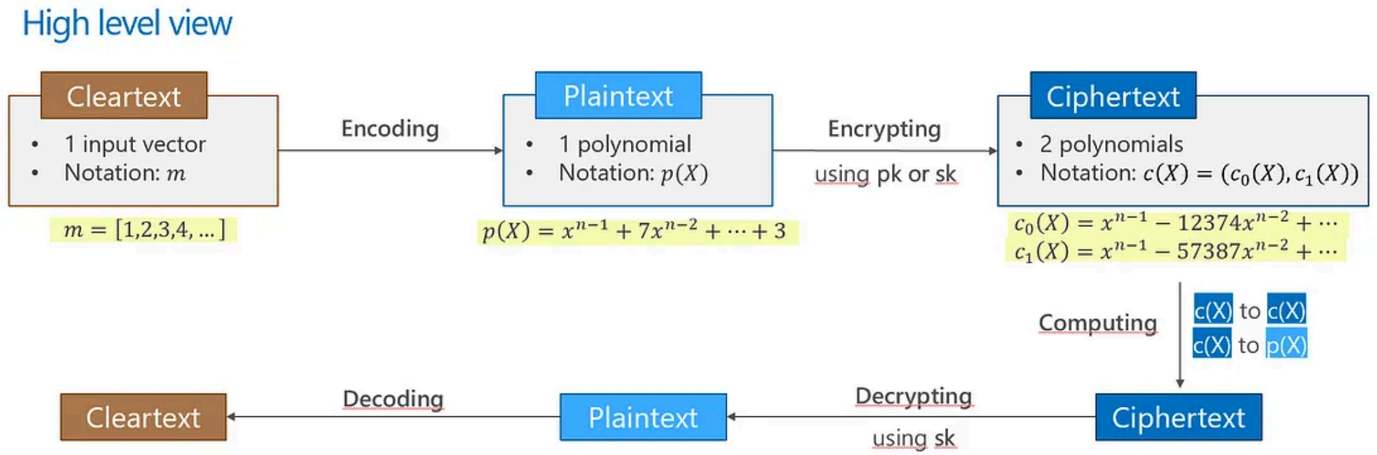


(b) Encrypted image processing service model

Methodology

We will be using the CKKS scheme for FHE.

These schemes give us the capability of performing addition, multiplication and subtraction on encrypted data.



Our plan - To encrypt each pixel individually and do computation on them.

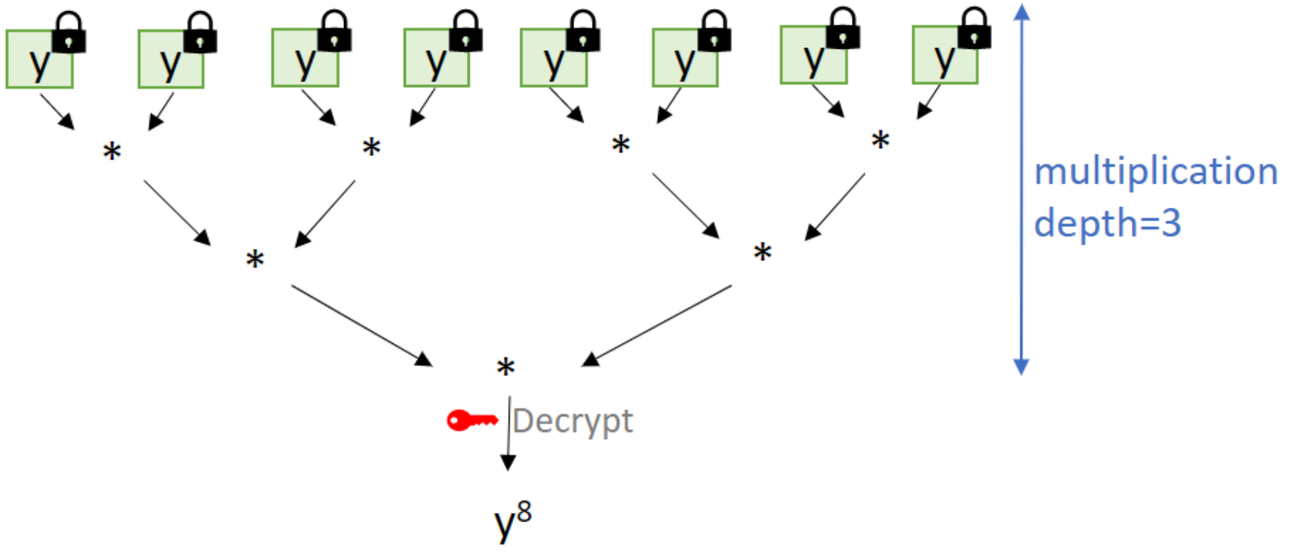
Why CKKS?

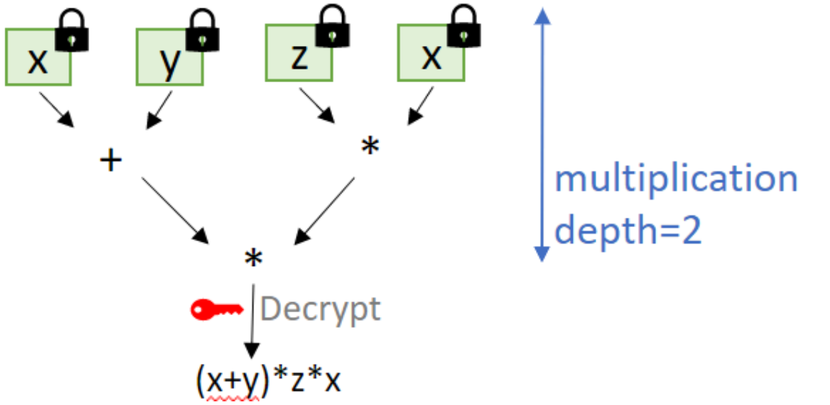
Why we need to use floating point representation?

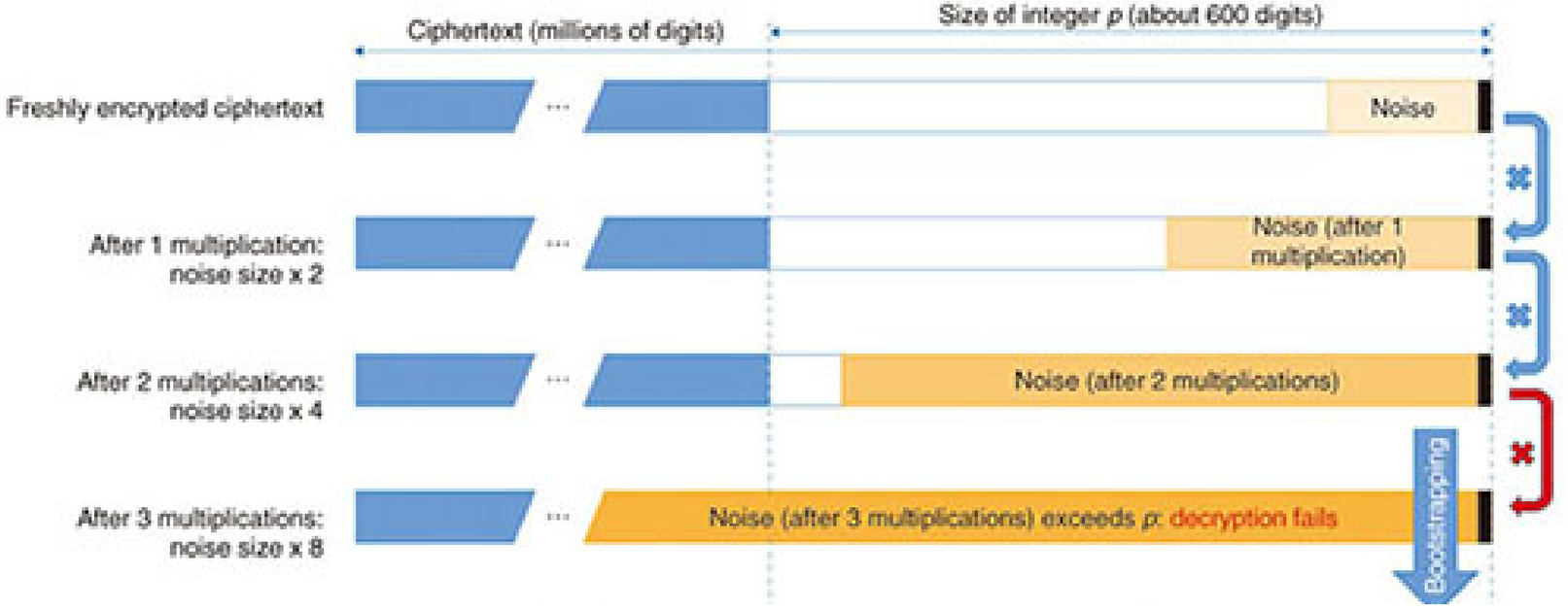
- We have operations like Gradients and Hessian

Limitations and Constraints of CKKS -

- Multiplicative Depth
- Noise


y
y
y
y
y
y
y
y
*
*
*
*
*
*
*
Decrypt
y8
multiplication
depth=3


x
y
z
x
+
*
*
Decrypt
(x+y)*z*x
multiplication
depth=2


Ciphertext (millions of digits)
Size of integer p (about 600 digits)
Freshly encrypted ciphertext
Noise
After 1 multiplication:
noise size x 2
Noise (after 1
multiplication)
After 2 multiplications:
noise size x 4
Noise (after 2 multiplications)
After 3 multiplications:
noise size x 8
Noise (after 3 multiplications) exceeds p: decryption fails
Bootstrapping

Breaking Down SIFT

Scale-space Extrema Detection

Goal : Identify points that are scale-invariant by searching for local extrema in the scale space

- Steps:
  - Create a scale-space by convolving the image with Gaussian filters at multiple scales.
  - Subtract adjacent Gaussian-blurred images to produce Difference of Gaussians (DoG).
  - Find local extrema in the DoG images by comparing each pixel with its neighbors in both the spatial (3x3 region) and scale dimensions.

- Computational Tasks:
  - Convolve the image with Gaussian kernels (computationally expensive).
  - Subtract Gaussian images to create the DoG images.
  - Iterate through the image and scales to find local extrema.

Keypoint Localization

Goal : Refine the location of detected keypoints and remove unstable ones

- Steps:
  - Use interpolation (e.g., Taylor expansion) to refine keypoint locations.
  - Discard keypoints with low contrast or those lying on edges (unstable keypoints).
  - Low contrast: Magnitude of DoG value is below a threshold.
  - Edge response: Use the Hessian matrix's eigenvalues to determine edge-like regions and discard them.

- Computational Tasks:
  - Interpolate extrema locations to sub-pixel accuracy.
  - Compute Hessian matrices and eigenvalues for edge detection.

Orientation Assignment

Goal : Assign an orientation to each keypoint for rotation invariance

- Steps:
  - Compute gradient magnitudes and orientations for pixels around the keypoint.
  - Create an orientation histogram of these gradients (e.g., 36 bins covering 360°).
  - Assign the dominant orientation (peak of the histogram) to the keypoint. Additional keypoints can be created for significant secondary peaks.

- Computational Tasks:
  - Compute gradient magnitude and direction for surrounding pixels.
  - Construct and analyze the orientation histogram.

Keypoint Descriptor Generation

Goal : Generate a descriptor vector that describes the local image region around the keypoint in a robust and invariant way.

- Steps:
  - Take a 16x16 neighborhood around the keypoint.
  - Divide the neighborhood into a 4x4 grid of 4x4 subregions.
  - Compute gradient magnitudes and orientations for each subregion.
  - Create an 8-bin orientation histogram for each subregion.
  - Concatenate these histograms to form a 128-dimensional feature vector.

- Computational Tasks:
  - Compute gradients and orientations for the 16x16 neighborhood.
  - Build orientation histograms for each subregion.
  - Normalize the descriptor vector for illumination invariance.

The Easy Parts

The Primitives

```
def secAdd(a, b):
    return a + b
```

```
def secSub(a, b):
    return -b + a #
```

```
def secMul(a, b):
    return a * b
```

Resizing

```
def secResize(image, dsize, fx=None, fy=None):
    if fx is None and fy is None:
        fx = dsize[0] / image.shape[1]
        fy = dsize[1] / image.shape[0]

    new_image = np.zeros((dsize[1], dsize[0]), dtype=ts.CKKSVector)
    for y in range(dsize[1]):
        for x in range(dsize[0]):
            new_image[y, x] = image[int(y / fy), int(x / fx)]

    return new_image
```

Convolution

```
    for i in range(img_height):
        for j in range(img_width):
            patch = img_padded[i:i + kernel_height, j:j + kernel_width]
            vector_prod = sec2DVectorProduct(patch, kernel)
            img_out[i, j] = sec2DVectorSum(vector_prod)

    return img_out
```

Gaussian Blurr

```
def secGaussianBlur(image, kernel_size, sigma):
    kernel = np.zeros((kernel_size, kernel_size))
    center = kernel_size // 2
    for i in range(kernel_size):
        for j in range(kernel_size):
            kernel[i, j] = np.exp(-((i - center) ** 2 + (j - center) ** 2) / (2 * sigma ** 2))
    kernel /= kernel.sum()

    new_image = secConvolve2d(image, kernel)
    return new_image
```

Difference of Gaussians

```
def secGenerateDoGImages(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_in_octave[1:]):
            dog_images_in_octave.append(secSubtract2DVector(second_image, first_image))
        dog_images.append(dog_images_in_octave)
    return dog_images
```

LSTSQ and Inverse Matrix

```python
def secLTSQ(X, y):
    X = np.array(X)
    y = np.array(y)
    X_transpose = np.transpose(X)

    XTX = np.dot(X_transpose, X)
    XTy = np.dot(X_transpose, y)

    XTX_inv, denominator = inv_3x3(XTX)

    beta = np.dot(XTX_inv, XTy)
    return beta, denominator
```

```python
def inv_3x3(x):
    a, b, c = x[0]
    d, e, f = x[1]
    g, h, i = x[2]
    det = a * (e * i - f * h) - b * (d * i
    det_inv = 1
    return np.array([[det_inv * (e * i - f
                      [det_inv * (f * g - d
                      [det_inv * (d * h - e
```

The Hard Parts

Comparision

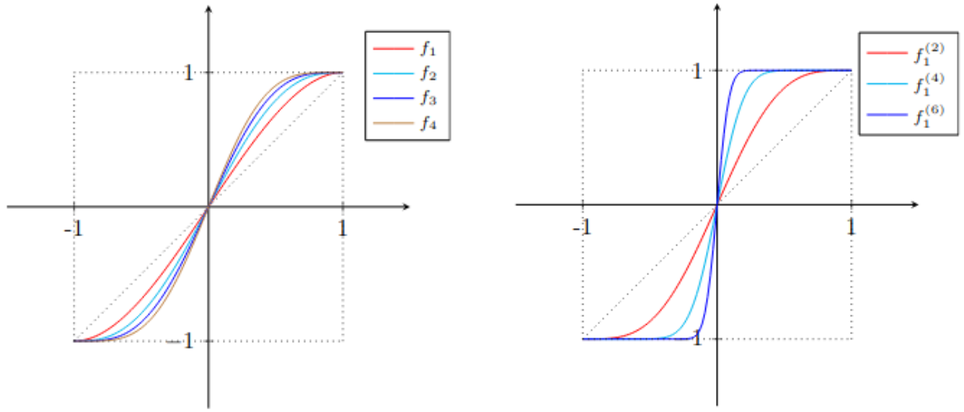
A challenging task due to the complexity of implementing comparison functions.

$$f_n(x) = \sum_{i=0}^{n} \frac{1}{4^i} \cdot \binom{2i}{i} \cdot x(1-x^2)^i.$$

Algorithm 1 `NewComp`$(a, b; n, d)$

Input: $a, b \in [0, 1]$, $n, d \in \mathbb{N}$
Output: An approximate value of 1 if $a > b$, 0 if $a < b$ and $1/2$ otherwise

1: $x \leftarrow a - b$
2: **for** $i \leftarrow 1$ **to** d **do**
3: $\quad x \leftarrow f_n(x)$ // compute $f_n^{(d)}(a-b)$
4: **end for**
5: **return** $(x+1)/2$



(a) f_n for $n = 1, 2, 3, 4$ (b) $f_1^{(d)}$ for $d = 2, 4, 6$

Fig. 3: Illustration of $f_n^{(d)}$

More Practical Solution :- A practical solution involves an interactive approach where the server sends the two numbers to the client for comparison, and the client returns an encrypted boolean result with minimal noise

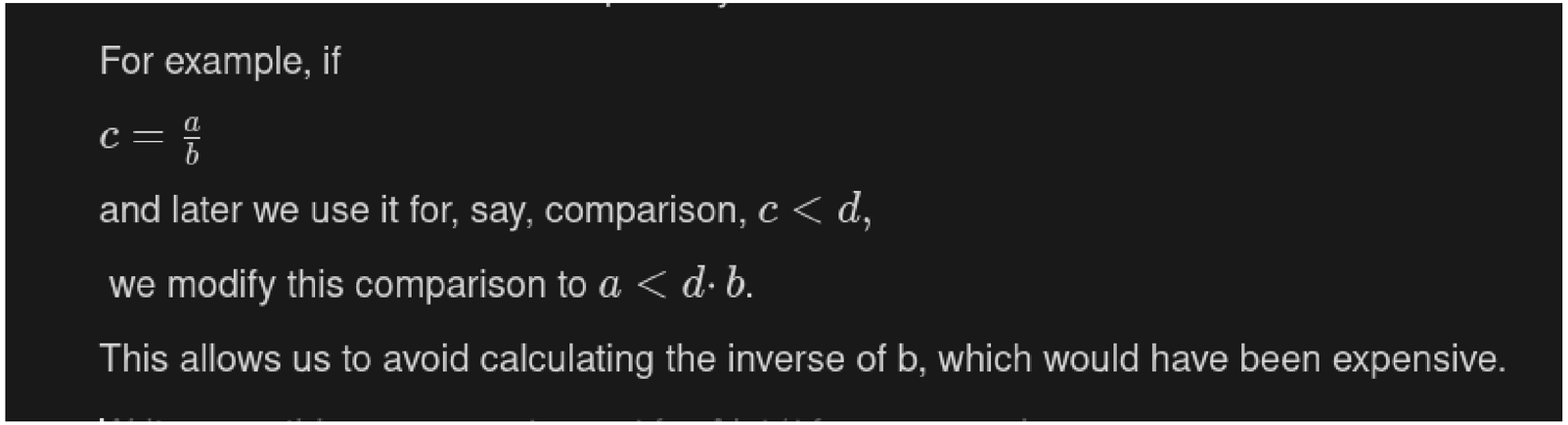
```
def main():
    # Client Side
    comm = Comm()
    comm.start_client()
    context, secret_key = init_enc()

    def cmp(x, a, b):
        x = dill.loads(x)
        a = dill.loads(a)
        b = dill.loads(b)
        if isinstance(x, ts.CKKSVector):
            x = x.decrypt(secret_key)[0]
        if isinstance(a, ts.CKKSVector):
            a = a.decrypt(secret_key)[0]
        if isinstance(b, ts.CKKSVector):
            b = b.decrypt(secret_key)[0]
        return dill.dumps(ts.ckks_vector(context=context, vector=[int(a < x < b)]))
```

Division

- Integer division is surprisingly hard to perform using FHE primitives, it often requires the use comparison function which itself is expensive.
- Floating point division is a bit simpler; approximations such as Taylor or Chebyshev series are commonly used in FHE computations To get a good approximation we still need to use excessive multiplicative depth.

$$\frac{1}{x} \approx \frac{1}{a} - \frac{(x-a)}{a^2} + \frac{(x-a)^2}{a^3} - \frac{(x-a)^3}{a^4} + \cdots$$

A work around - We store the numerator and denominator of the division separately and modify the subsequent uses of the algorithm. To use the numerator and denominator separately



This allows us to avoid calculating the inverse of b, which would have been expensive.

Function Approximation

There are steps in SIFT which require the computation of the magnitude of vectors. This involves computing the square root of a number. Again there are polynomial approximations of this using Chebyshev polynomials (used by openFHE).

Any sort of polynomial approximation would require a lot of multiplicative depth to get a good approximate value, so we do the same that we did in comparison and delegate this calculation to the client.

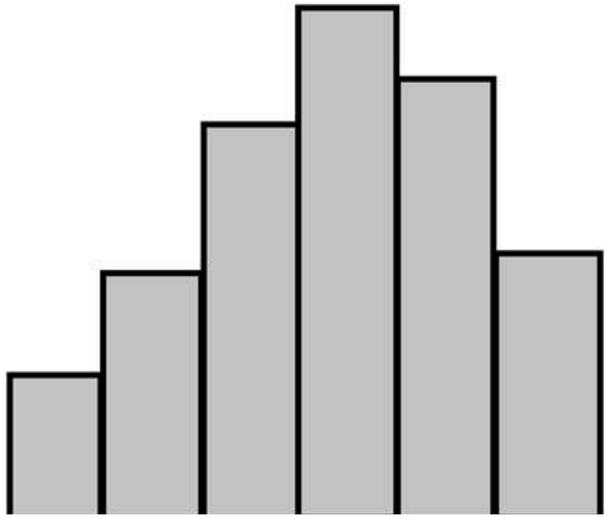
Handling Conditional Blocks

The result of a comparison is often used to decide which code block to execute. In algorithms implemented with FHE primitives, we do not have a luxury to choose.

If the boolean value of an if else condition is encrypted then we need to execute both the if-block and the else-block and mask the effects of the block by multiplying with the boolean value.

```
for something:
        if condition :
                continue
        #expensive computation
```

Histograms and Binning



For a given angle, one would calculate the index of the array that needs to be incremented.

- Use a Mask - for every i

$$a_i < \theta < a_{i+1}$$

Do the comparision as follows, since finding theta is hard

$$\theta = \arctan\left(\frac{dy}{dx}\right)$$

$$\tan(a_i) \cdot dx < dy < \tan(b_i) \cdot dx$$

Note all a_i are not encrypted since they are just the bin boundries

Array Max

The usual method of doing this would be maintaining a running maximum of the elements and comparing that running maximum with subsequent elements.

```
max = 0
for element in array:
  b = element > max
  max = b * element + (1 - b) * max
```

If we analyze the multiplicative depth of this algorithm, it would be O(N), where N is size of array, ignoring the requirement for comparison.

One way around -
Since length of arrays can be large, we instead use a different way of calculating max, where we compare adjacent elements of the array and remove the smaller values. This would cut down the array size in half. We perform this step till only one element is left.

```
def vecmax(ls):
  l = len(ls)
  if l == 1:
      return ls[0]
  elif l == 2:
      return max(ls[0], ls[1])
  else:
      return max(vecmax(ls[:l//2]), vecmax(ls[l//2:]))
```

```
def max(a, b):
  cond = a > b
  return cond * a + (1-cond) * b
```

The multiplicative depth of this algorithm ignoring comparison is O(logN).

Deferred Computation

In order to delegate expensive operations, such as comparison and square root, to the client in a non-interactive manner, we propose a novel method of deferring computation.

Consider the following expression:

$$a = (x > y) \cdot c + (z > w) \cdot d + (y \geq x) \cdot e$$

The server will return to the client:

$$f(c_1, c_2) = c_1 \cdot (c - e) + c_2 \cdot d + e$$

where:

$$c_1 = (x > y), \quad c_2 = (z > w)$$

In the Algorithm

Finding Space Scale Extrema

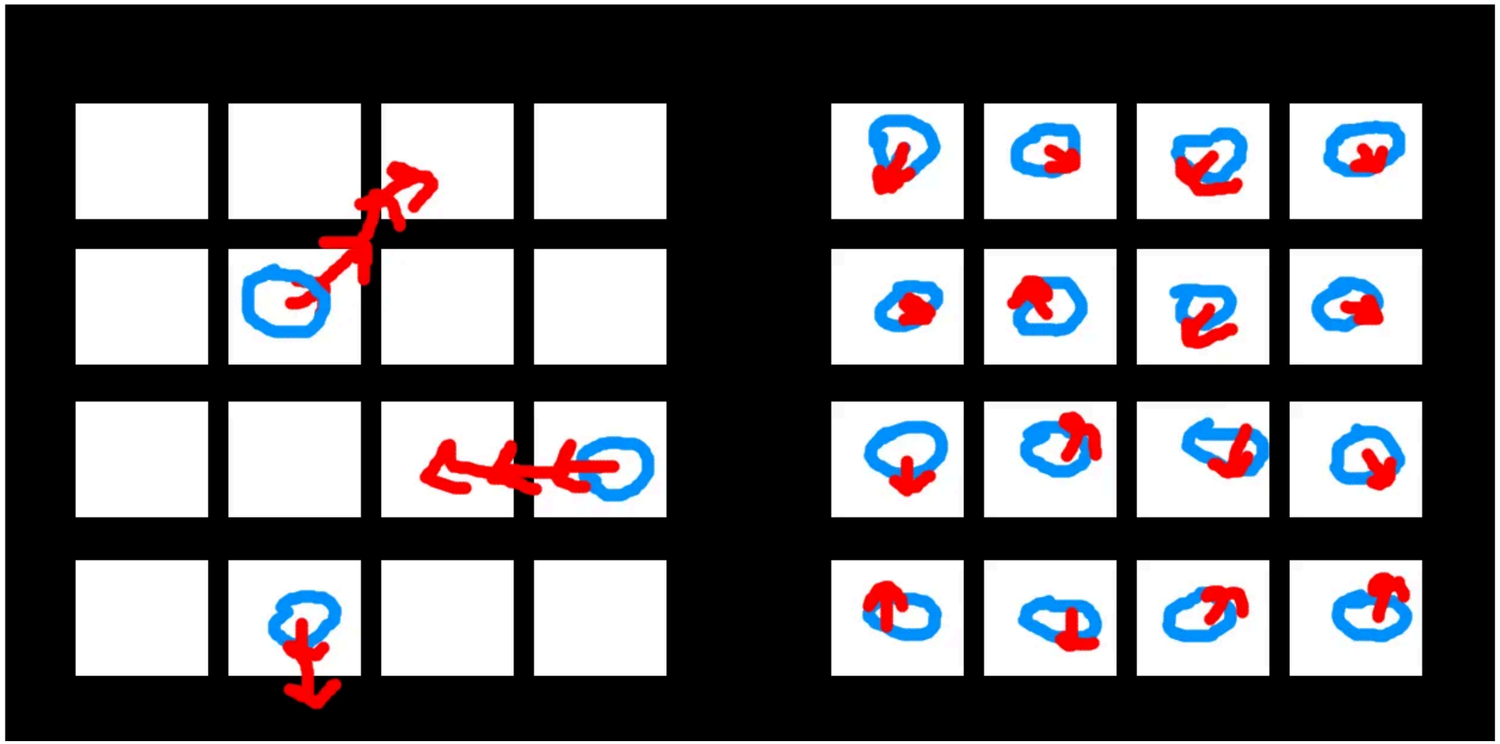
This is the original code -

```
    for octave_index, dog_images_in_octave in enumerate(dog_images):
        for image_index, (first_image, second_image, third_image) in enumerate(zip(dog_images_in_octave, do
            print("first_image shape: ", first_image.shape)
            # (i, j) is the center of the 3x3 array
            for i in range(image_border_width, first_image.shape[0] - image_border_width):
                for j in range(image_border_width, first_image.shape[1] - image_border_width):
                    if isPixelAnExtremum(first_image[i-1:i+2, j-1:j+2], second_image[i-1:i+2, j-1:j+2], thi
                        localization_result = localizeExtremumViaQuadraticFit(i, j, image_index + 1, octave
                        if localization_result is not None:
                            keypoint, localized_image_index = localization_result
                            keypoints_with_orientations = computeKeypointsWithOrientations(keypoint, octave
                            for keypoint_with_orientation in keypoints_with_orientations:
                                keypoints.append(keypoint_with_orientation)
    return keypoints
```

This is the our adapted code -

```
for octave_index, dog_images_in_octave in enumerate(dog_images):
    octave_keypoints = []
    for image_index, (first_image, second_image, third_image) in enumerate(zip(dog_images_in_oct
        # (i, j) is the center of the 3x3 array
        img_keypoints = []
        for i in range(image_border_width, first_image.shape[0] - image_border_width):
            row_keypoints = []
            for j in range(image_border_width, first_image.shape[1] - image_border_width):
                keypoint = localizeExtremumViaQuadraticFit(i, j, image_index + 1, octave_index,
                keypoints_with_orientations = computeKeypointsWithOrientations(keypoint, octave_
                flat_list.append(keypoints_with_orientations)
                row_keypoints.append(keypoints_with_orientations)
            img_keypoints.append(row_keypoints)
        octave_keypoints.append(img_keypoints)
    keypoints.append(octave_keypoints)
return keypoints, flat_list
```

What allows us to skip the expensive gradient ascent computation in the neighbouring pixels.



A function is fit on the values of pixels in that neighbourhood and hessian and a method similar to gradient ascent is used to calculate the maxima point.
In our setting - The gradient ascent now doesn’t need to go beyond the current pixel’s influence to determine whether to remove or keep the current pixel. We can get away with using only one gradient ascent step. This fact is illustrated in the figure above.

The original code

```
def localizeExtremumViaQuadraticFit(i, j, image_index, octave_index, num_intervals, dog_images_in_octave, si
    """Iteratively refine pixel positions of scale-space extrema via quadratic fit around each extremum's ne
    """
    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 rescale pixel values to [
        first_image, second_image, third_image = dog_images_in_octave[image_index-1:image_index+2]
        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]
        # print("extremum_update: ", extremum_update)
        if abs(extremum_update[0]) < 0.5 and abs(extremum_update[1]) < 0.5 and abs(extremum_update[2]) < 0.5
            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
```

Our Adaptation

```
def localizeExtremumViaQuadraticFit(i, j, image_index, octave_index, num_intervals
    """Iteratively refine pixel positions of scale-space extrema via quadratic fit
    """
    print("Num attempts: ", num_attempts_until_convergence)
    extremum_update = np.array([1, 1, 1], dtype='float32')  # (di, dj, ds)
        # need to convert from uint8 to float32 to compute derivatives and need to
    first_image, second_image, third_image = dog_images_in_octave[image_index-1:im
    pixel_cube = np.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]]) * (1 / 255)
    gradient = computeGradientAtCenterPixel(pixel_cube)
    hessian = computeHessianAtCenterPixel(pixel_cube)
    ltsq_val, denominator = secLTSQ(hessian, gradient)
    # ltsq_val = ltsq_val[0]
    extremum_update = -ltsq_val
    extremum_update = refresh(extremum_update)
```

We retain three checks to determine if a point is a keypoint -

- Extrema Identification
  - Extrema Updated calculated from LSTSQ fitting over hessian and gradient should be less than < 0.5
- Contrast Threshold
  - We check if the maxima/extrema pixel has high enough from its surrounding
- Eliminating edge maximas
  - We estimate the curvature at points with 2 x 2 hessian at the point
  - r is the ratio of these eigen-values determined by the hessian
    - $r > 10$, we eliminate the keypoint as a flatter curvature means a straight line

Keypoint with Orientation

We introduce our own keypoint class. response and is_key_point are encrypted here.

```
class EncKeyPoint:
    def __init__(self, i, j, octave, is_keypoint_present, size, response, angle=None):
        self.i = i
        self.j = j
        self.octave = octave
        self.is_keypoint_present = is_keypoint_present
        self.size = size
        self.response = response
        self.angle = angle
```

```python
def find_angle(dx, dy, num_bins=360, cmp=None):
    tan_right_bins = np.zeros(num_bins // 4) * dx    # when angle is -45 to 45 and the (135 to -135, anticlock
    tan_left_bins = np.zeros(num_bins // 4) * dx     # when angle is -45 to 45 and the (135 to -135, anticlock
    cot_up_bins = np.zeros(num_bins // 4) * dx       # otherwise, abs(dx) < abs(dy)
    cot_down_bins = np.zeros(num_bins // 4) * dx      # otherwise, abs(dx) < abs(dy)

    # print("tan_right_bins : ", tan_right_bins +1)
    np.cot = lambda x: -np.tan(x + np.pi/2)

    tan_right_edges = np.linspace(-45, 45, num_bins // 4 + 1)
    tan_left_edges = np.linspace(135, 225, num_bins // 4 + 1)
    cot_up_edges = np.linspace(45, 135, num_bins // 4 + 1)
    cot_down_edges = np.linspace(225, 315, num_bins // 4 + 1)

    for i, (l, r) in enumerate(zip(tan_right_edges, tan_right_edges[1:])):
        cond = cmp(dy, dx * np.tan(np.deg2rad(l)), dx * np.tan(np.deg2rad(r)))
        tan_right_bins[i] += cond
```

Our binning strategy needs us to take tan values of all the boundary angles.

If the angles we are comparing with are close to 90 then the tan value becomes close to infinity. This would make the comparison unreliable.

Therefore for the parts of comparison where we have angles that give tan values more than 1, we use cot instead and modify the inequality to

```
dy * cot (a_i) < dx < dy * cot(a_i+1)
```

Keypoint Descriptors

In this stage, we use the same trick as before for histogram binning to calculate descriptors
However this time the angle is with respect to keypoint orientation and the number of bins are different

We also need to calculate the distance of the angle from the center of each of the 8 orientations
in order to calculate relative magnitude, since we don’t know which bin the orientation lies in,
we need to calculate for all and have a boolean mask to selectively update the bin magnitudes.

Results

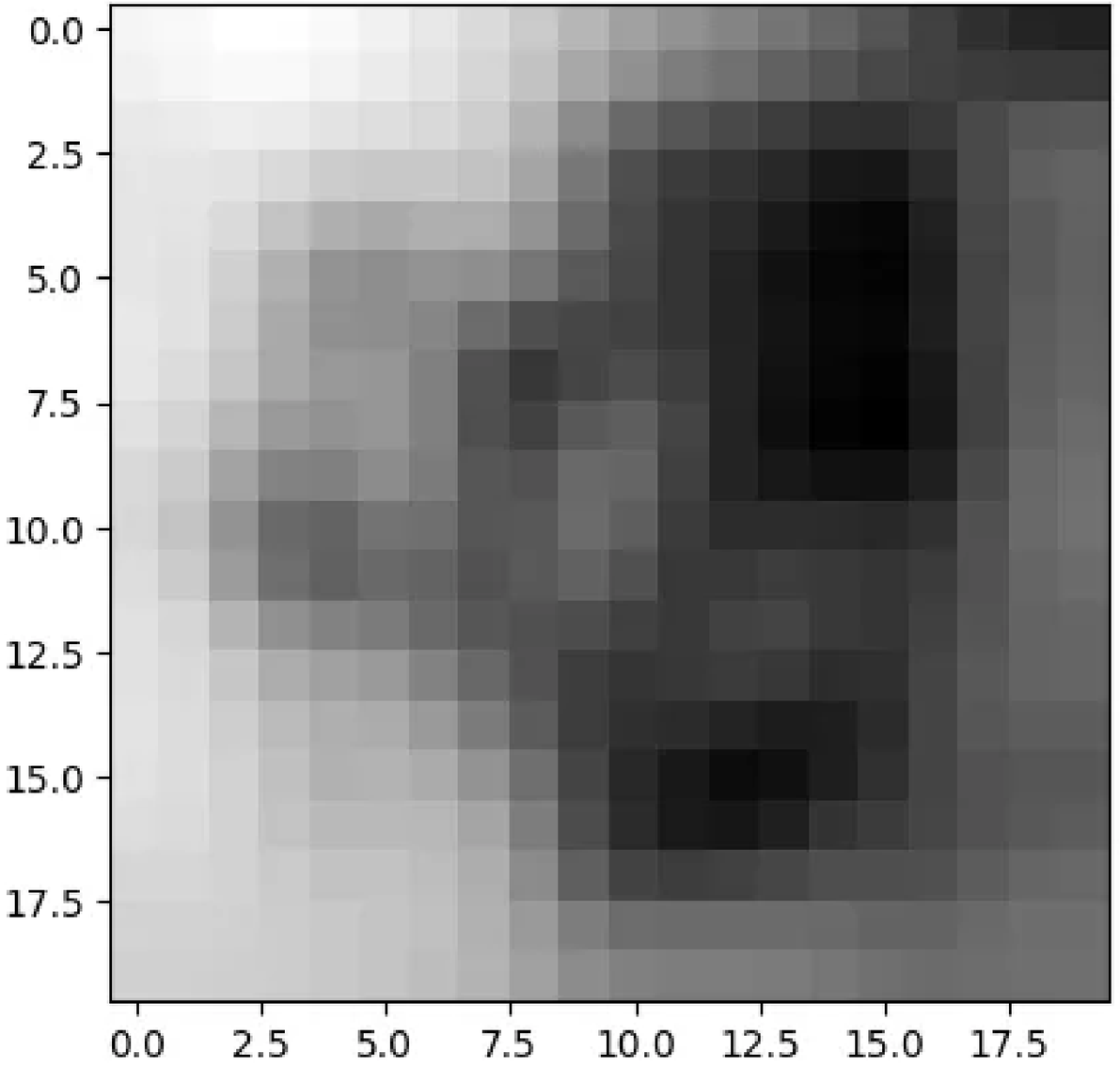
Time Analysis

Time Analysis for a 20 x 20 grey scale image.

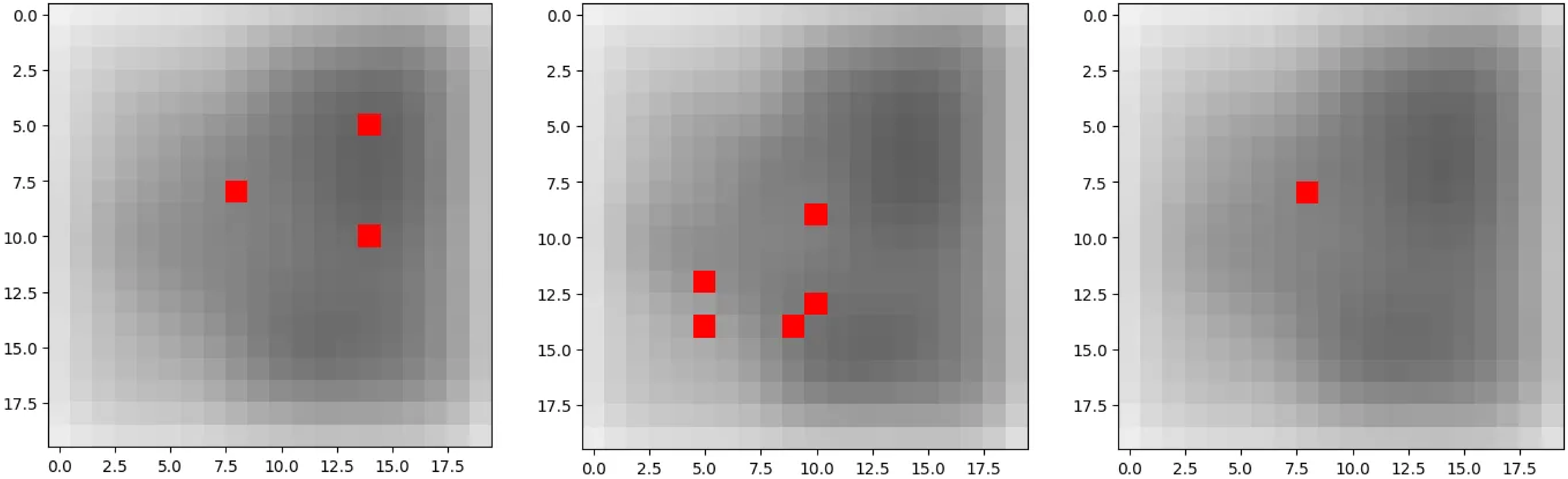
- Generating Base Image - 19.5 seconds
- Num Octaves - 0.0 seconds
- gaussian_kernel_sizes - 0.0 seconds
- Gaussian Images - 6 minute 56.7 seconds
- dog_images - 2.5 seconds
- keypoints -
- findScaleSpaceExtrema
    - Orientation - 1 keypoint takes 2 minutes
- Descriptors - 1 keypoint takes 5 minutes

Example Image

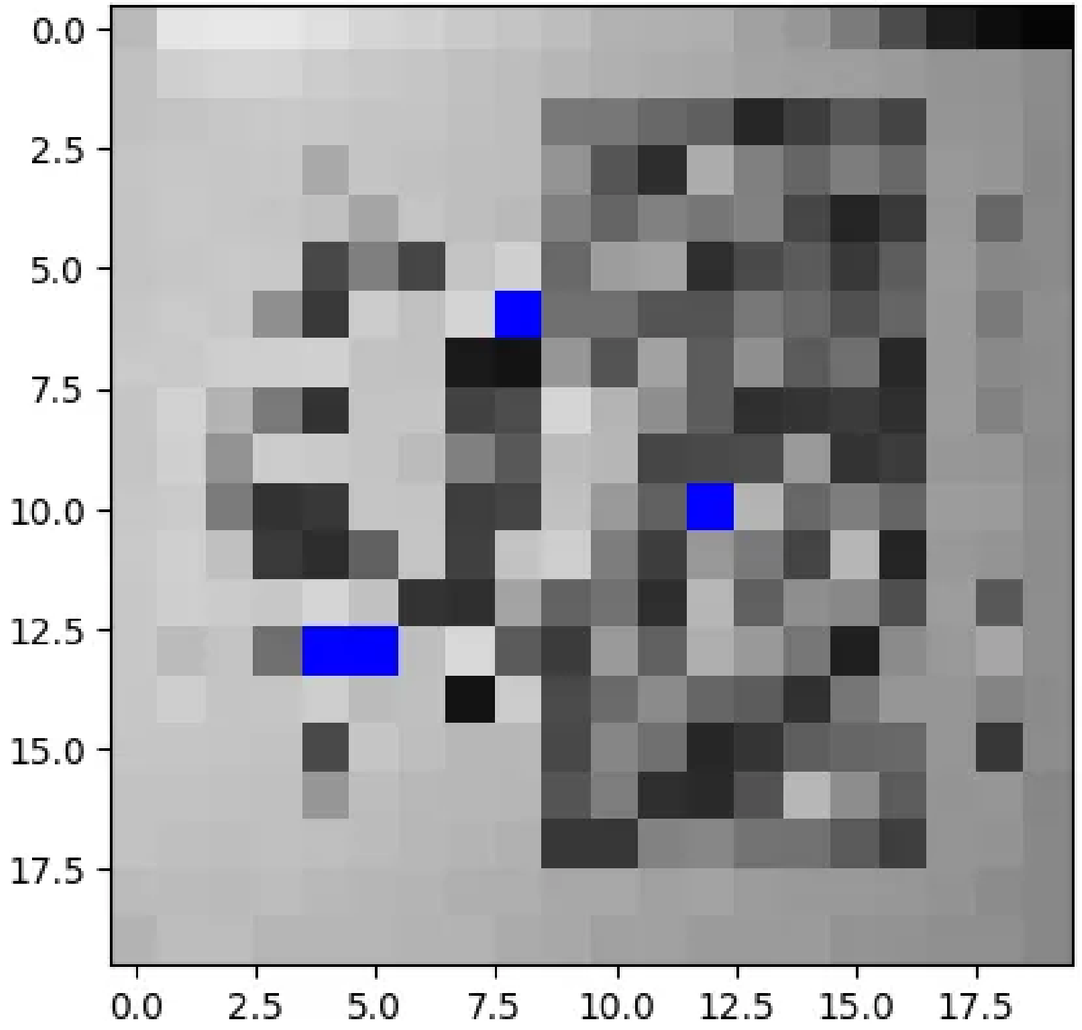
Resized and Blurred Base Image



Keypoints plotted on its respective DoG image

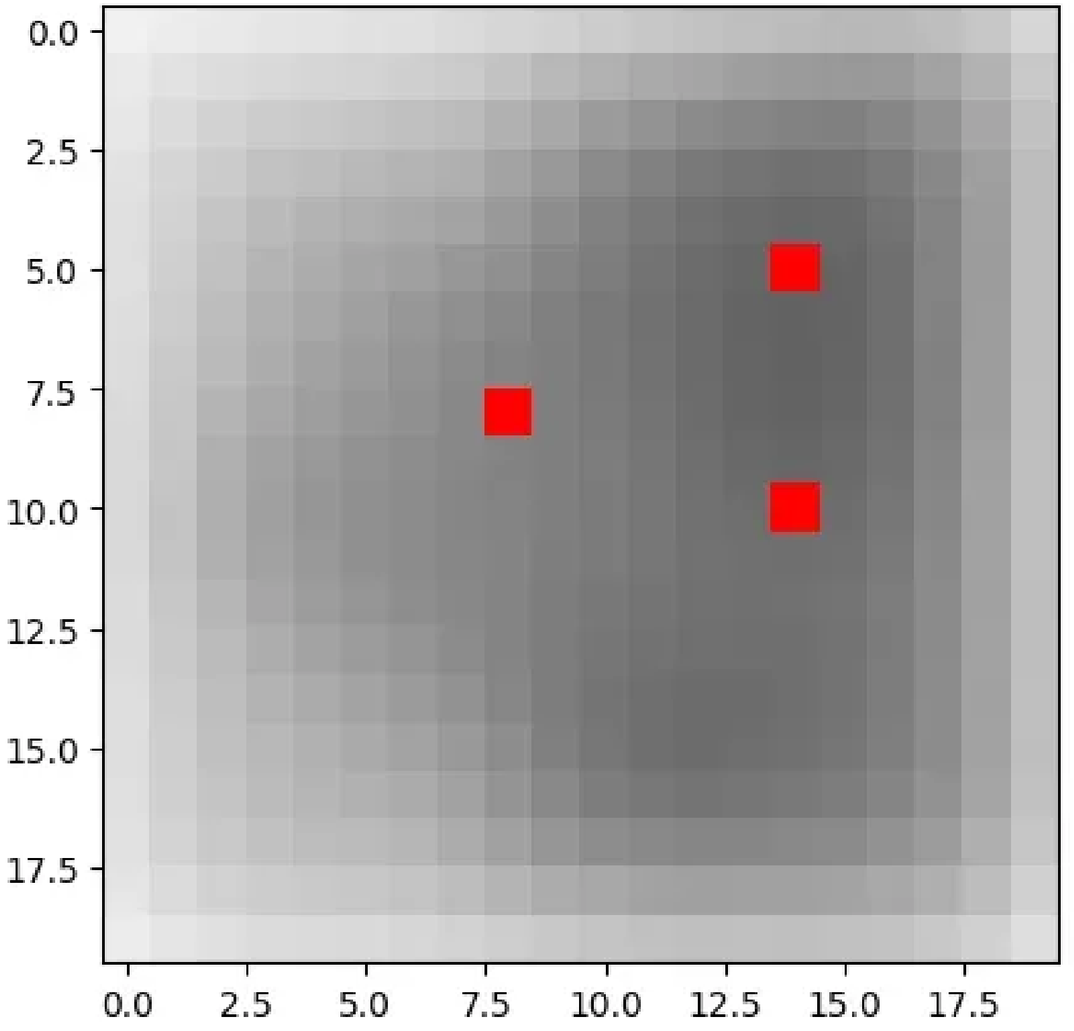


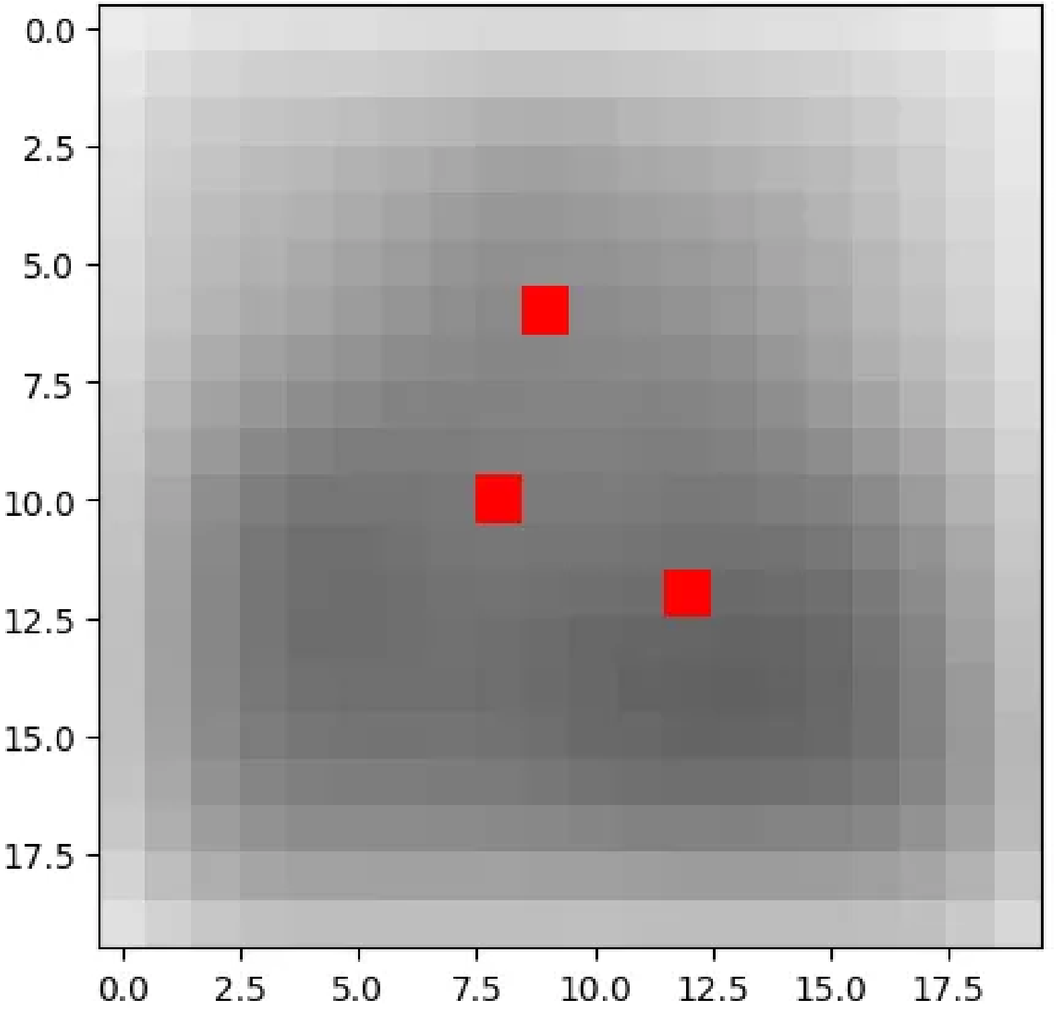
Keypoints from a normal SIFT algorithm (plotted on the actual image)

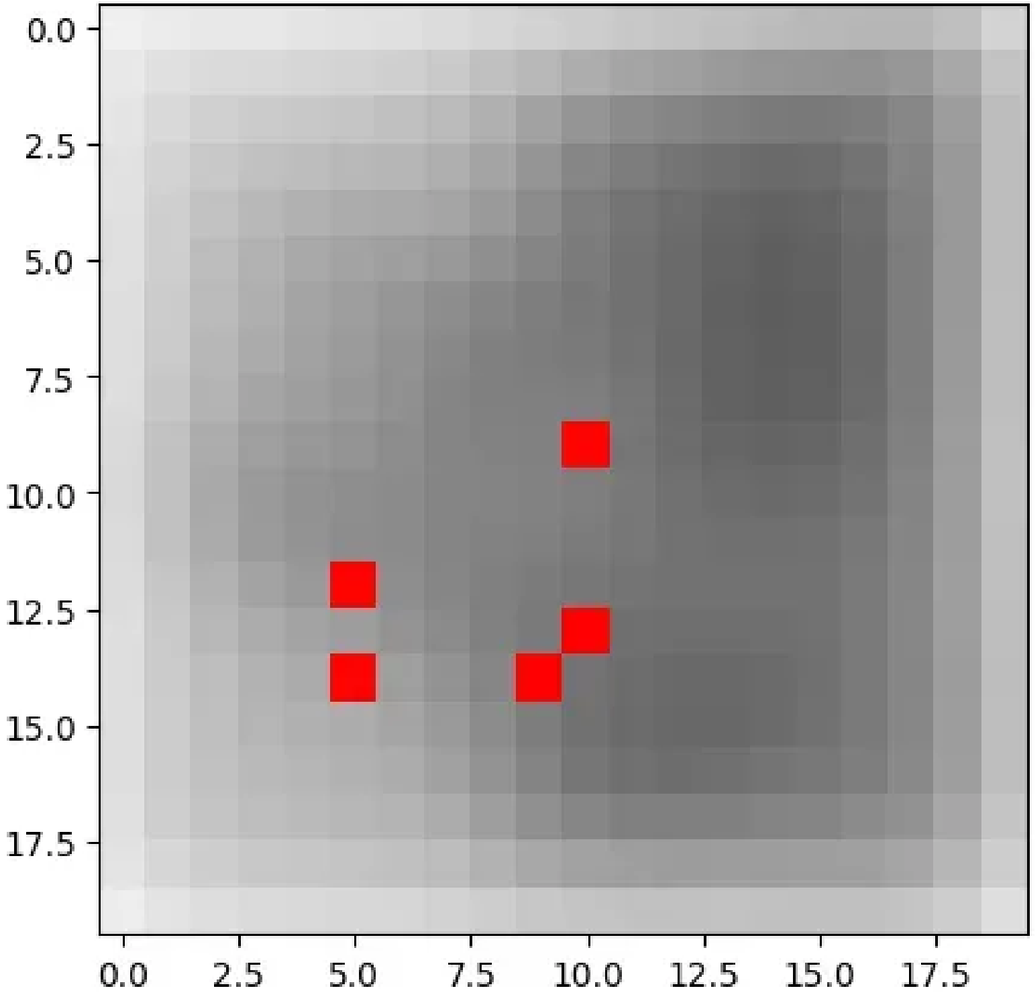


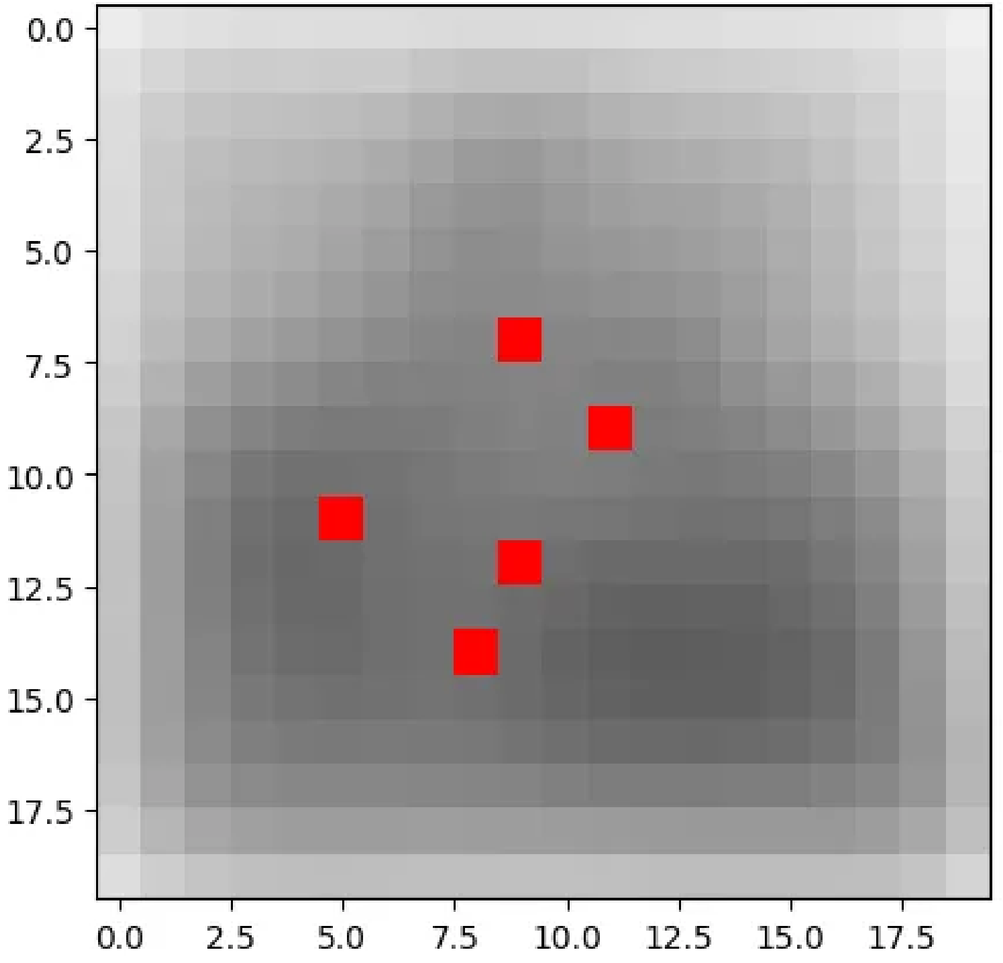
Verifying Correctness

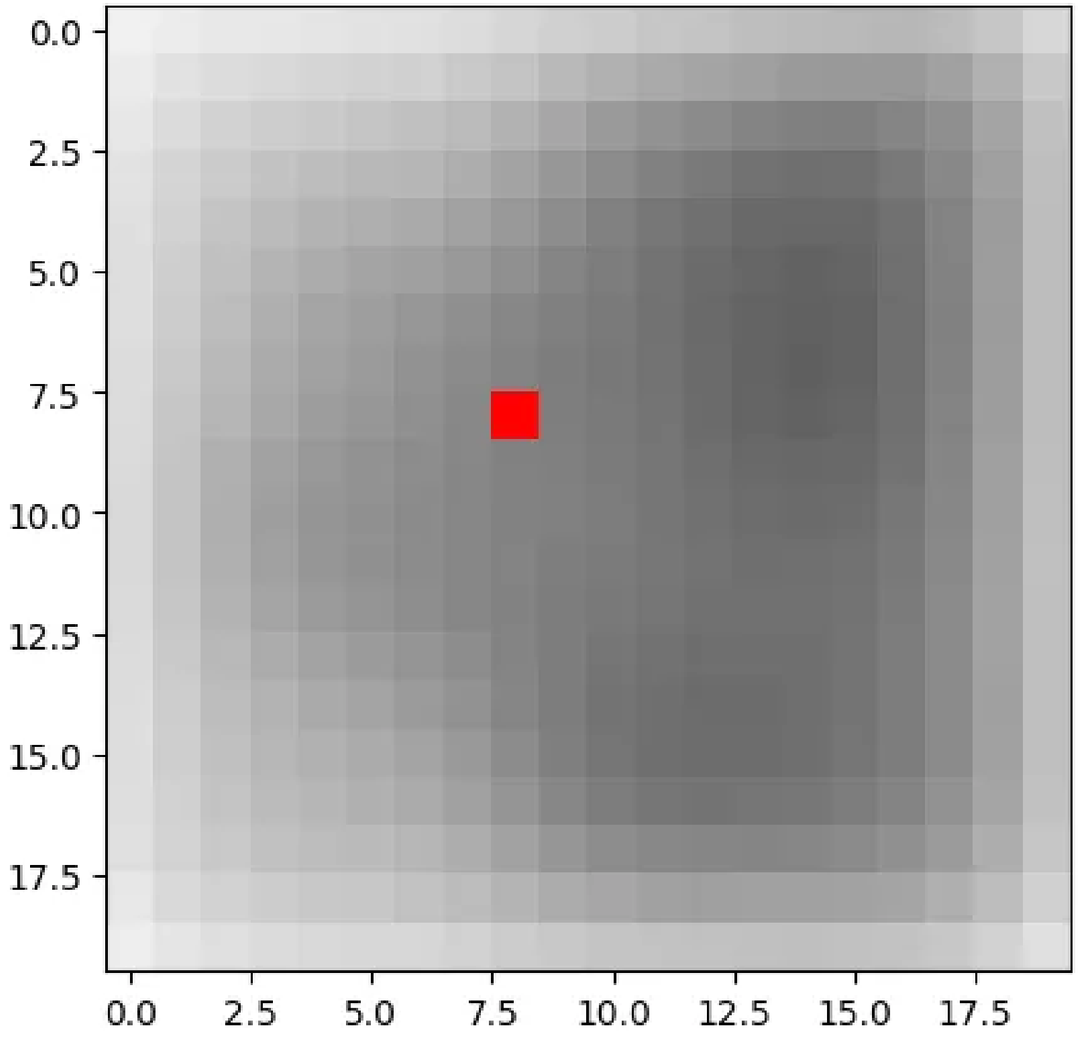
If the keypoints our algorithm is generating are invariant to changes (say rotation), that means they are indeed SIFT points. We do that check to verify our correctness

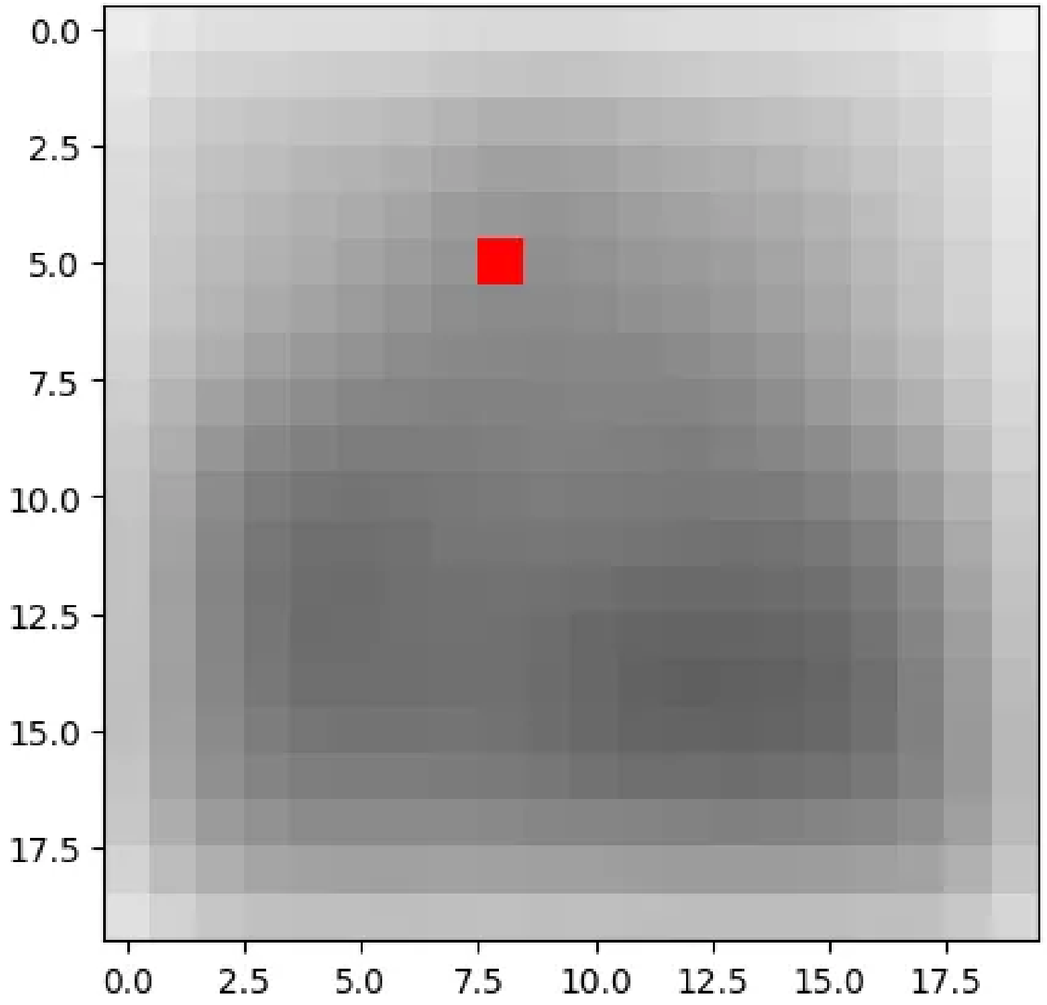

0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

Future Work

Parallelization

- Scope of optimizing convolution operations.
- Computation for each pixel is independent at the key point finding stage
- Same is the case for every keypoint while finding orientation and descriptors
- Parallelization here will significantly decrease
- Use DFT,DTFT or FFT for making the convolution step faster

More efficient Deferred Computation

- Can delegate more items and not just comparisons for deferred computations.
- Parser to compress the computation graph generated as a result of minimize algorithm leak and computation.
- PyTorch’s Compiler can be used to optimize the computational graph that would sent towards the end.
- Checkout src/secsift/defer.py for the initial prototype of the computation graph.

General Image Processing in FHE

- This opens a pandora’s box, any image processing algorithm can be moved to the encryption domain.
- Descriptors of different images can be compared to check/compared in the encrypted domain.
- Apart from image comparision, things like image stitching can also be done
- Solves the problem of remote computation mentioned in the problem statement.

A Practical Exercise in Adapting SIFT Using FHE Primitives

Ishwar B Balappanawar*, Bhargav Srinivas Kommireddy*
{ishwar.balappanawar, bhargav.srinivas}@students.iiit.ac.in
IIIT Hyderabad
* *These authors contributed equally to this work.*

***Abstract*—An exercise in implementing Scale Invariant Feature Transform using CKKS Fully Homomorphic encryption quickly reveals some glaring limitations in the current FHE paradigm. These limitations include the lack of a standard comparison operator and certain operations that depend on it (like array max, histogram binning etc). We also observe that the existing solutions are either too low level or do not have proper abstractions to implement algorithms like SIFT.**

In this work, we demonstrate:

- **Methods of adapting regular code to the FHE setting.**
- **Alternate implementations of standard algorithms (like array max, histogram binning, etc.) to reduce the multiplicative depth.**
- **A novel method of using deferred computations to avoid performing expensive operations such as comparisons in the encrypted domain.**

Through this exercise, we hope this work acts as a practical guide on how one can adapt algorithms to FHE.

I. Introduction

How good is the current Fully Homomorphic Encryption (FHE) paradigm at performing common everyday algorithms? To

- Keypoint Localization
- Orientation Assignment
- Keypoint Descriptor Generation

Our goal was to implement all these computational tasks using the primitives provided to us for a FHE scheme. Some of the tasks involve computing gradients and hessians which warrant the use of floating point numbers. For this reason we chose to use the CKKS encoding scheme[10]. We used the tenseal python library, as it is a well supported wrapper around the popular Microsoft SEAL library. The primitives that we utilize are addition and multiplication. So we implemented all the necessary computational tasks using addition and multiplication. There is the further constraint of multiplicative depth, which prevents us from performing a long chain of multiplications on an encrypted number. This is one of the most significant hurdles with adapting any algorithm to the FHE setting.

Some of the tasks, such as convolution and matrix multiplication,

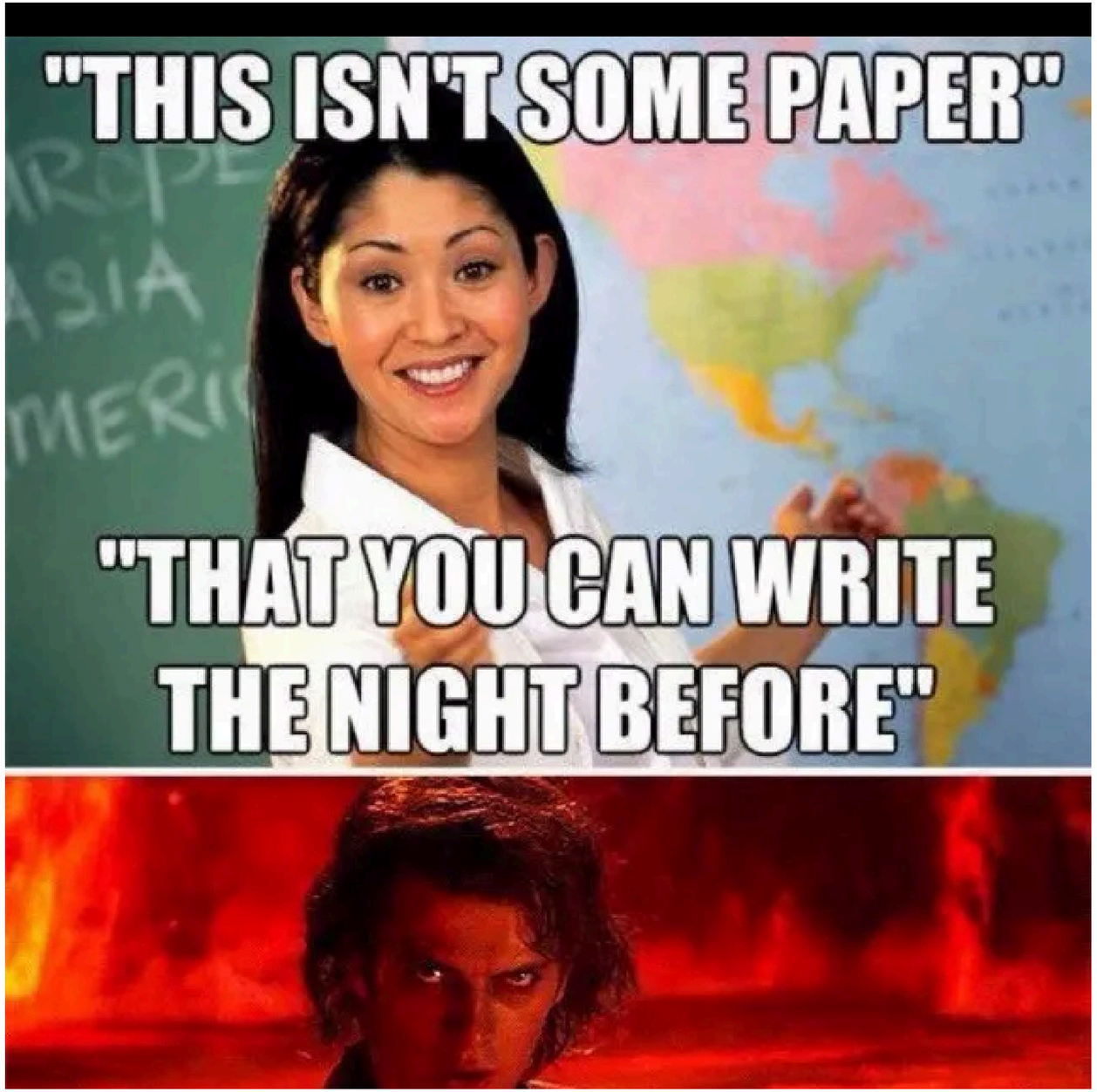
We have submitted a version of this work to FHE.org conference which will be colocated with Real World Crypto 2025

References

[1] T. Lindeberg, Scale Invariant Feature Transform, 05 2012, vol. 7.
[2] M. Naehrig, K. Lauter, and V. Vaikuntanathan, “Can homomorphic encryption be practical?” Proceedings of the 3rd ACM Workshop on Cloud Computing Security, pp. 113–124, 2010.
[3] S. Chhabra and et al., “A survey on fully homomorphic encryption: An engineering perspective,” IEEE Access, vol. 8, pp. 217 050–217 073, 2020.
[4] Microsoft, “Seal (simple encrypted arithmetic library),” 2017. [Online]. Available: https://github.com/microsoft/SEAL
[5] S. Halevi and V. Shoup, “Helib,” 2014. [Online]. Available: https://github.com/homenc/HElib
[6] T. Contributors, “Tenseal: A library for encrypted tensor computation,” 2020. [Online]. Available: https://github.com/OpenMined/TenSEAL
[7] C.-Y. Hsu, C.-S. Lu, and S.-C. Pei, “Image feature extraction in encrypted domain with privacy-preserving sift,” IEEE transactions on image processing : a publication of the IEEE Signal Processing Society, vol. 21, pp. 4593–607, 06 2012.
[8] X. Liu, X. Zhao, Z. Xia, Q. Feng, P. Yu, and J. Weng, “Secure outsourced sift: Accurate and efficient privacy-preserving image sift feature extraction,” IEEE Transactions on Image Processing, vol. 32, pp. 4635–4648, 2023.

[9] I. Rey Otero and M. Delbracio, “Anatomy of the SIFT Method,” Image Processing On Line, vol. 4, pp. 370–396, 2014, https://doi.org/10.5201/ipol.2014.82.
[10] J. H. Cheon, A. Kim, M. Kim, and Y. Song, “Homomorphic encryption for arithmetic of approximate numbers,” in Advances in Cryptology – ASIACRYPT 2017, T. Takagi and T. Peyrin, Eds. Cham: Springer International Publishing, 2017, pp. 409–437.
[11] W. Fu, R. Lin, and D. Inge, “Fully homomorphic image processing,” 10 2018.
[12] J. H. Cheon, A. Kim et al., “Comparison algorithms for homomorphic encryption,” Cryptology ePrint Archive, vol. 2017, p. 1164, 2017. [Online]. Available: https://eprint.iacr.org/2017/1164
[13] D. Bootland, I. Iliashenko, and C. Martindale, “Optimal polynomial approximations for modular exponentiation,” Cryptology ePrint Archive, vol. 2020, p. 1213, 2020.
[14] G. S. Brodal, “Cache-oblivious algorithms and data structures,” Lecture Notes in Computer Science, vol. 2497, pp. 268–286, 2001.
[15] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, A. Ravi, and J. McMahan, “Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy,” International Conference on Machine Learning, pp. 201–210, 2016.

[16] H. Qin and et al., “Privacy-preserving sift: A secure framework for image feature extraction,” IEEE Transactions on Information Forensics and Security, vol. 13, no. 8, pp. 2068–2084, 2018.
[17] M. Abadi and et al., “Tensorflow: A system for large-scale machine learning,” Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 265–283, 2016.
[18] C. Marcolla, V. Sucasas, M. Manzano, R. Bassoli, F. H. Fitzek, and N. Aaraj, “Survey on fully homomorphic encryption, theory, and applications,” Cryptology ePrint Archive, Paper 2022/1602, 2022. [Online]. Available: https://eprint.iacr.org/2022/1602
[19] O. D. Team, “Openfhe library,” 2021. [Online]. Available: https://github.com/openfheorg/openfhe-developmen


"THIS ISN'T SOME PAPER"
"THAT YOU CAN WRITE THE NIGHT BEFORE"