Task 1

```
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     4                                           def res_skimage2(imgs):
     5         1      32326.0  32326.0     35.9      imgs_float = imgs.astype(float) / 255.0
     6
     7         1         60.0     60.0      0.1      new_size = (imgs.shape[1] // 2, imgs.shape[2] // 2)
     8
     9         1      56415.0  56415.0     62.6      res_im = transform.downscale_local_mean(imgs_float, (1, 2, 2))
    10
    11         1       1252.0   1252.0      1.4      res_im_uint8 = np.asarray(res_im * 255, dtype=np.uint8)
    12
    13         1          7.0      7.0      0.0      return res_im_uint8

Original code execution time: 0.477856 seconds
Updated code execution time: 0.022650 seconds
```

This was the bottleneck point in the code. I altered the code to convert it into float data type and am performing a transform.downscale_local_mean() operation. This is a functionality of the skimage.transform. It reducing the spatial resolution of an image while preserving local mean information

Task 2

```
It took 0.22996759414672852 secs for 1822725 iterations
It took 2.788308620452881 secs for 22059421 iterations
It took 4.0476155281066895 secs for 32374695 iterations
It took 11.38483738899231 secs for 88754320 iterations
It took 1.2170300483703613 secs for 9716266 iterations
It took 25.419209718704224 secs for 200745654 iterations
Sequential Execution Time: 45.09372663497925 seconds
It took 0.28484582901000977 secs for 1822725 iterations
It took 1.5922002792358398 secs for 9716266 iterations
It took 3.6021604537963867 secs for 22059421 iterations
It took 4.965150594711304 secs for 32374695 iterations
It took 12.553197860717773 secs for 88754320 iterations
It took 26.809941053390503 secs for 200745654 iterations
The program took: 27.25656247138977 secs
Parallel Execution Time: 27.2645742893219
```

I have implemented multiprocessing as this task can readily be divided into multiple processes without an hassle. Avoiding the Global Interpreter Lock in such CPU-bound scenario also played a role in using multiprocessing over multithreading. Parallel execution sped up the task by around 45%.

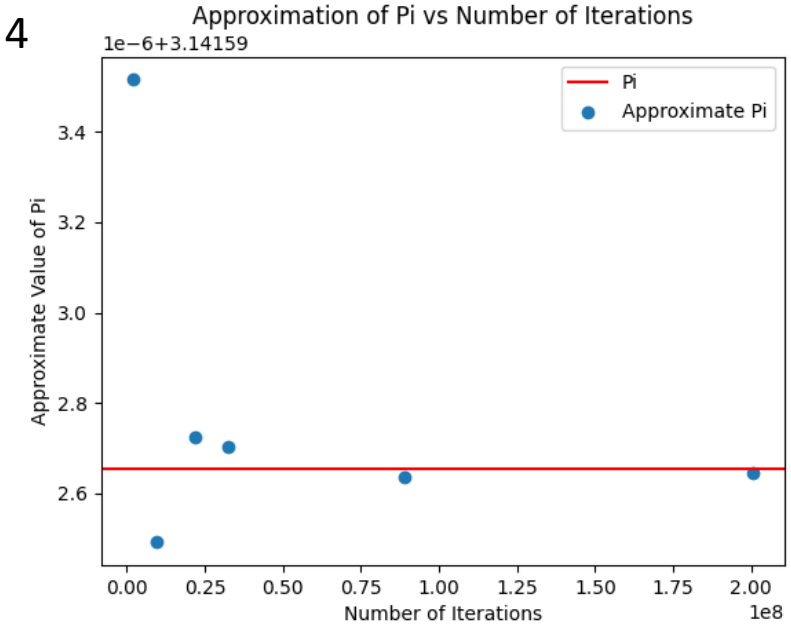
Task 3

```python
import time
from multiprocessing import Pool, cpu_count
from numba import jit

@jit(nopython=True)
def approximate_pi(n):
    pi_2 = 1
    nom, den = 2.0, 1.0
    for i in range(n):
        pi_2 *= nom / den
        if i % 2:
            nom += 2
        else:
            den += 2
    return 2 * pi_2

if __name__ == "__main__":
    nums = [1_822_725, 22_059_421, 32_374_695, 88_754_320, 97_162_66, 200_745_654]
    start_time_sequential = time.time()
    results_sequential = [approximate_pi(n) for n in nums]
    end_time_sequential = time.time()
    total_sequential_execution_time = end_time_sequential - start_time_sequential
    print(f"Sped up Execution Time: {total_sequential_execution_time} seconds")
```

```
Sped up Execution Time: 1.612074851989746 seconds
```

By using @jit(nopython=True), the code sped up from 45.1 seconds to 1.6 seconds which is a major boost.

Task 4



Alap Mundayoor, M.N. 23351429, IdM: ow07ysym