图像视频编码大实验

陈嘉杰 2017011484

代码采用 Python 编写，依赖 Pillow numpy matplotlib scipy opencv2 等库

Exp1 Are they equivalent in effect?

子任务1 转换为灰度图片

见代码 `grayscale.py`，直接采用 PIL 的相关函数即可。

原图：

灰度：

子任务2 尝试用不同方式对图片进行 DCT

代码在 `lena_dct_exp1.py` 中。

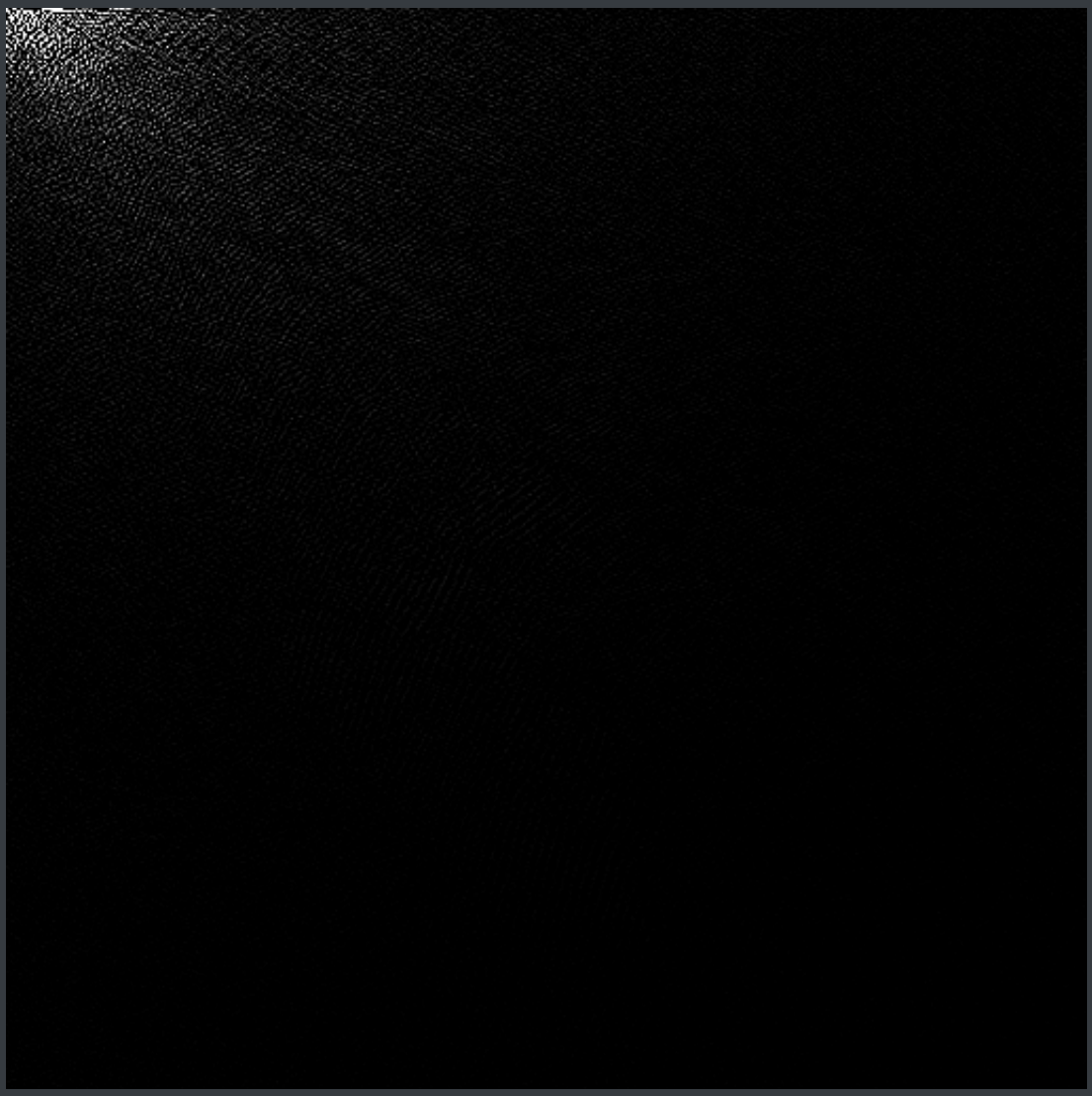
第一个方法是，先对行再对列进行 DCT ，第二个方法是，对整个图进行 DCT，这两个方法在数学上是等价的，只不过在后面 1/4 1/16 和 1/64 时权值的选取上可以有不一样的结果，后面会继续讨论。

第三个方法则是分割成 8x8 以后进行 DCT。

相关代码：

```
def dct2d(data):
    return fftpack.dct(fftpack.dct(data, norm='ortho').T, norm='ortho').T

def idct2d(data):
    return fftpack.idct(fftpack.idct(data, norm='ortho').T, norm='ortho').T
```

第一个和第二个方法得到的DCT的图：



可以看到，左上角的数值是比较大的，其他地方的数都很小，比较符合 DCT 的特征。

切分为 8x8 以后也有类似的分布：

也是只有左上角一到两个像素比较大。

在运行时间上，设图片都是正方形，边长为 n ，那么第一种方法需要循环 n * n * n * 2 ，第二种方法需要 n * n * n * n 次，第三种方法需要 8 * 8 * 8 * 8 * (n/8) * (n / 8) = 64 * n * n ，当 n 比较大的时候第三种方法最快。

代码输出了对应的 PSNR 值。由于第一种方法和第二种方法在数学上是相等的，代码中只用了第一种方法进行计算，得到 PSNR 为 315.48 ，比第三种方法的 PSNR 315.45 略大，说明考虑到计算精度的时候，第一种方法比第三种方法能留下更精确的信息。

接着对 DCT 之后的系数进行了“压缩”，题目要求 1/4 1/16 和 1/64 ，首先对 8*8 的格子进行了 DCT 系数的选取，方法是，如果是 1/4 ，则选取左上角的 4*4，剩下为零，其它依此类推。再用 IDCT 恢复`到原来的图像，相关代码：

```
def matrix_select(data, side):
    x, y = data.shape
    result = np.zeros(data.shape)
    for i in range(int(x/side)):
        for j in range(int(y/side)):
            result[i,j] = data[i,j]
    return result

```

对比如下：

直接还原：

1/4 的情况：

1/16 的情况：

1/64 的情况：

可以看到，随着压缩率不断增加，图片清晰度也逐渐下降，但仍然保留了比较多原始的信息。由于分块是按照 8x8 的，所以最后 1/64 比例时，每个 8*8 的块都是同一个像素值，显示出了明显的颗粒感。

直接对 2D DCT 进行类似的系数选取后，即对整个图片计算 2D DCT 后，保留左上角的一片系数，剩下都设置为 0，再 IDCT 恢复：

```
# 1/side^2 coefs
def full_compress(side):
    idct_4 = np.zeros(data.shape)
    dct_4 = matrix_select(dct,side)
    idct_4 = idct2d(dct_4)
    Image.fromarray(idct_4.clip(0,
255).astype('uint8')).save('lena_2ddct_%d_2didct.png' % (side ** 2))
    mse = np.mean((data - idct_4) ** 2)
    psnr = 10 * np.log10(255.0 ** 2 / mse)

full_compress(2) # 1/4
full_compress(4) # 1/16
full_compress(8) # 1/64
```

原始图片：

1/4 的情况：

1/16 的情况：

1/64 的情况：

可以看到，图片压缩率越高，清晰度也在不断下降，但是下降的形式和之前 8*8 时不大一样。因为是直接对全图的 DCT 系数进行压缩，所以在 1/64 的时候看到一些很明显的波纹，这些对应着留下来的部分 DCT 系数。

接下来是采用 PPT 文档中所描述的 DCT 系数选取方法，即先对行进行 DCT ，选取一半的列以后，对这一部分再进行 DCT ，剩余部分都为 0 。按照这样的策略，得到的图片为：

原始图片

1/4:

1/16:

1/64:

可以看到，也出现了一些比较明显的线条，和之前的结果类似。虽然操作顺序不同，但和之前 2D-DCT 最后取左上角的结果是一致的，所以最后得到的图片也是一样的，只是在运行时间上不一样而已。

Exp2 Why quantization is so important?

子任务 1 分块量化并计算平均 PSNR

代码在 `lena_dct_exp2.py` 中。

首先分块为 8x8 的小块，对每一块进行量化，代码如下：

```
def quantize(matrix, data, a):
    qq = a * matrix
    return np.round(data / qq) * qq
```

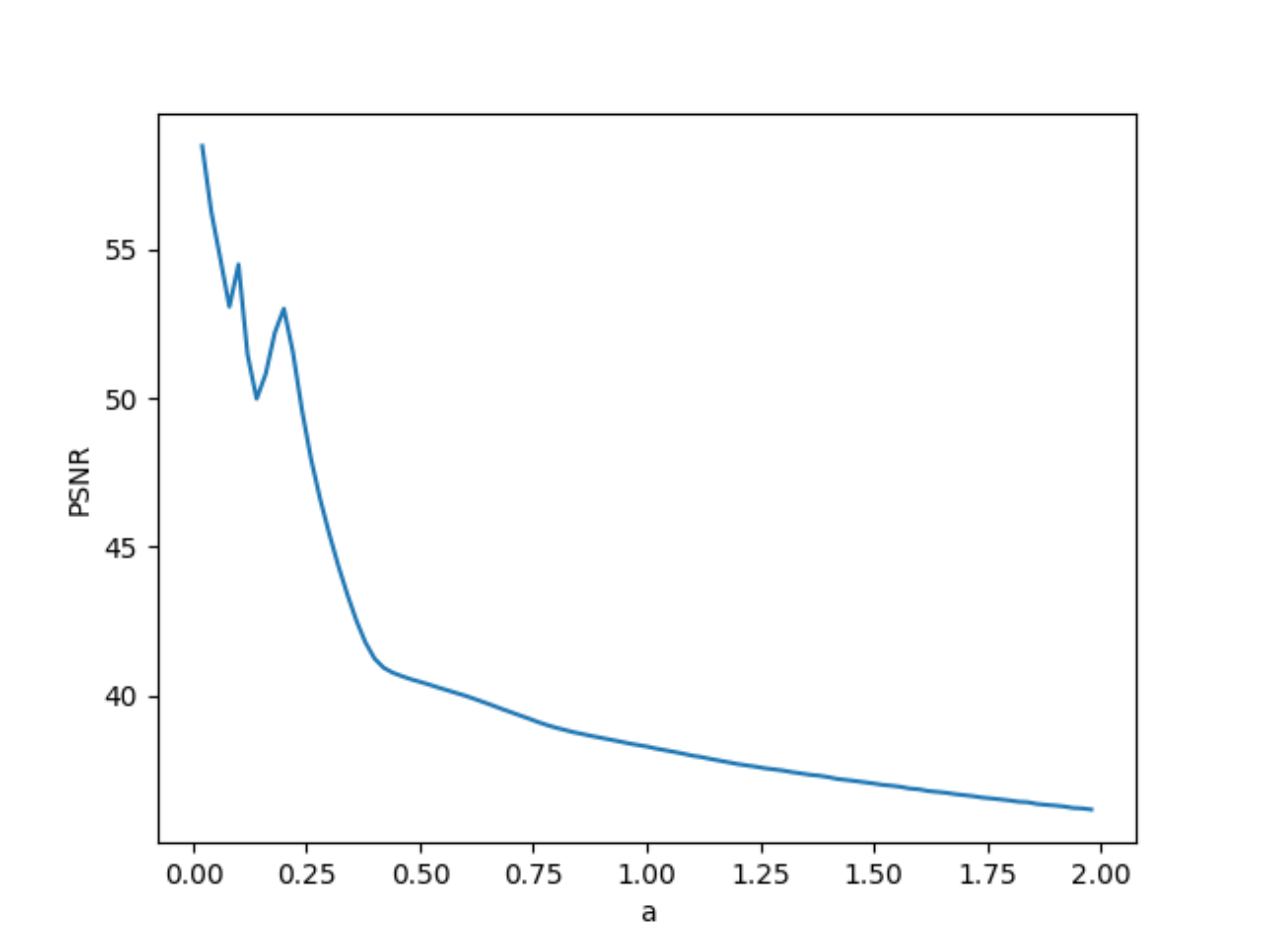
按照矩阵中对应的值，近似到最近的倍数上，a是Q的系数。通过计算，得到平均的 PSNR 为 38.28（a=1）时

子任务 2 根据不同的 a 得到 PSNR 曲线

代码：

```
for i in range(int(x/8)):
    for j in range(int(y/8)):
        submatrix = data[i*8:(i+1)*8, j*8:(j+1)*8]
        dct = dct2d(submatrix)

        for quan in range(1, 100):
            idct_quan = idct2d(quantize(Q, dct, quan / 50.0))
            psnr_8x8_quan[quan] += psnr(submatrix, idct_quan)
```

延续上面的思路，通过改变 a， 得到不同的 PSNR ，得到图如下：



可以看到，当 a 比较小的时候，此时量化矩阵的系数比较小，所以对原来的 DCT 系数矩阵的值的影响也比较小，所以大趋势是，随着a增大，PSNR减小，失真程度越高。有趣的是，在 a=0.10 和 a=0.20 出出现了两个小的尖峰，可能正好有一些数据在相邻的 a 值下量化到了同一个区间的两边，导致取值偏差较大。

接下来，找了一张图，测试 Canon 和 Nikon 的量化矩阵：

首先灰度处理：

接着按照类似的方法进行量化（代码在 `lena_dct_exp2_2.py` 中），得到：

```
psnr 2ddct 8x8 canon: 50.276670307604974
psnr 2ddct 8x8 nikon: 50.78738331730147
```

可以看到对于这个图片， Nikon 比 Canon 会稍微好一些。

对于量化矩阵的选取，可以看到它里面的数值有的大有的小，小则说明这个位置的 DCT 系数对视觉效果的影响比较大，反之说明影响比较小，量化以后可以得到比较高的压缩率，同时保证人眼看到的样子。所以左上角的数字一般比较小，右下角的数字一般比较大，这和 DCT 的意义是符合的。

写了简单的随机，来获得一个对于上面这个图片 PSNR 比较高的 Q 矩阵：

```
psnr 2ddct 8x8 best: 50.805403292236115
Q: [[ 1.  2.  1.  2.  3.  4.  6.  7.]
 [ 1.  1.  2.  3.  3.  6.  7.  7.]
 [ 2.  1.  2.  3.  4.  9.  8.  9.]
 [ 1.  1.  2.  4.  7. 13.  9.  8.]
 [ 3.  4.  6.  9.  9. 16. 13. 11.]
 [ 3.  4.  6.  8.  9. 12. 14. 10.]
 [ 7. 10. 11. 13. 14. 15. 15. 12.]
 [11. 13. 11. 14. 15. 15. 14. 11.]]
```

其效果也不是很好，并且也只能说明对于当前的这个图片，这个量化矩阵比较适合，但是不能保证它的普遍性，即对于各种图片都有比较好的效果。

Exp3 Intuitive interpretation of ME

子任务 1 选择 16*16 块并进行运动估计

代码在 `cars.py` 和 `cars_pixel.py` 中，前者负责把视频拆分成多个图片，后者负责追踪。

这次实验进行了两个物体的追踪，分别是白色的小轿车和白色的大巴车，它们的位置和出现的帧是手动标记的：

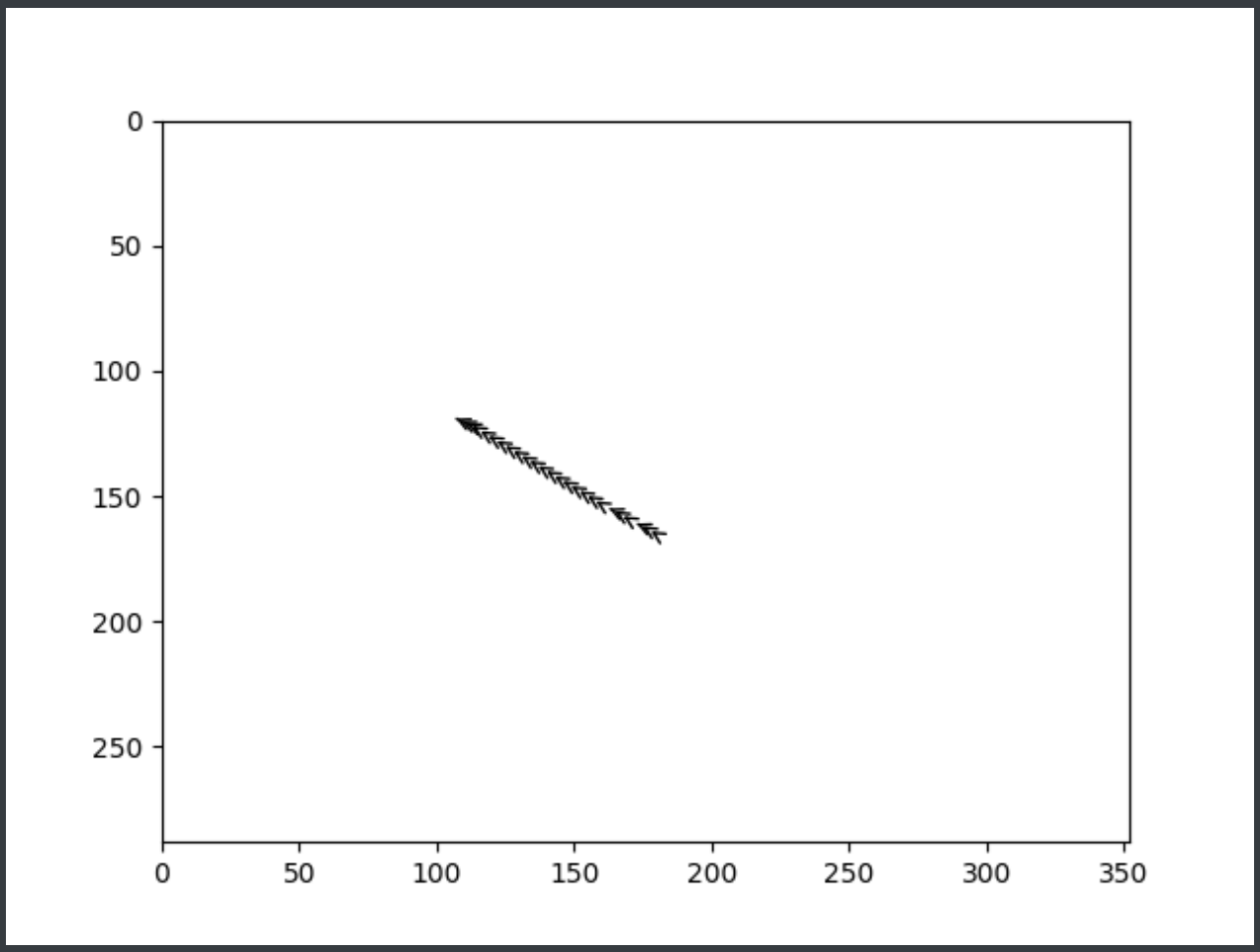
```
# white car in the middle
if sys.argv[1] == 'car':
    x = 166
    y = 181
    begin = 1
    end = 50

# bus in the right
else:
    x = 125
    y = 333
    begin = 10
    end = 170
```

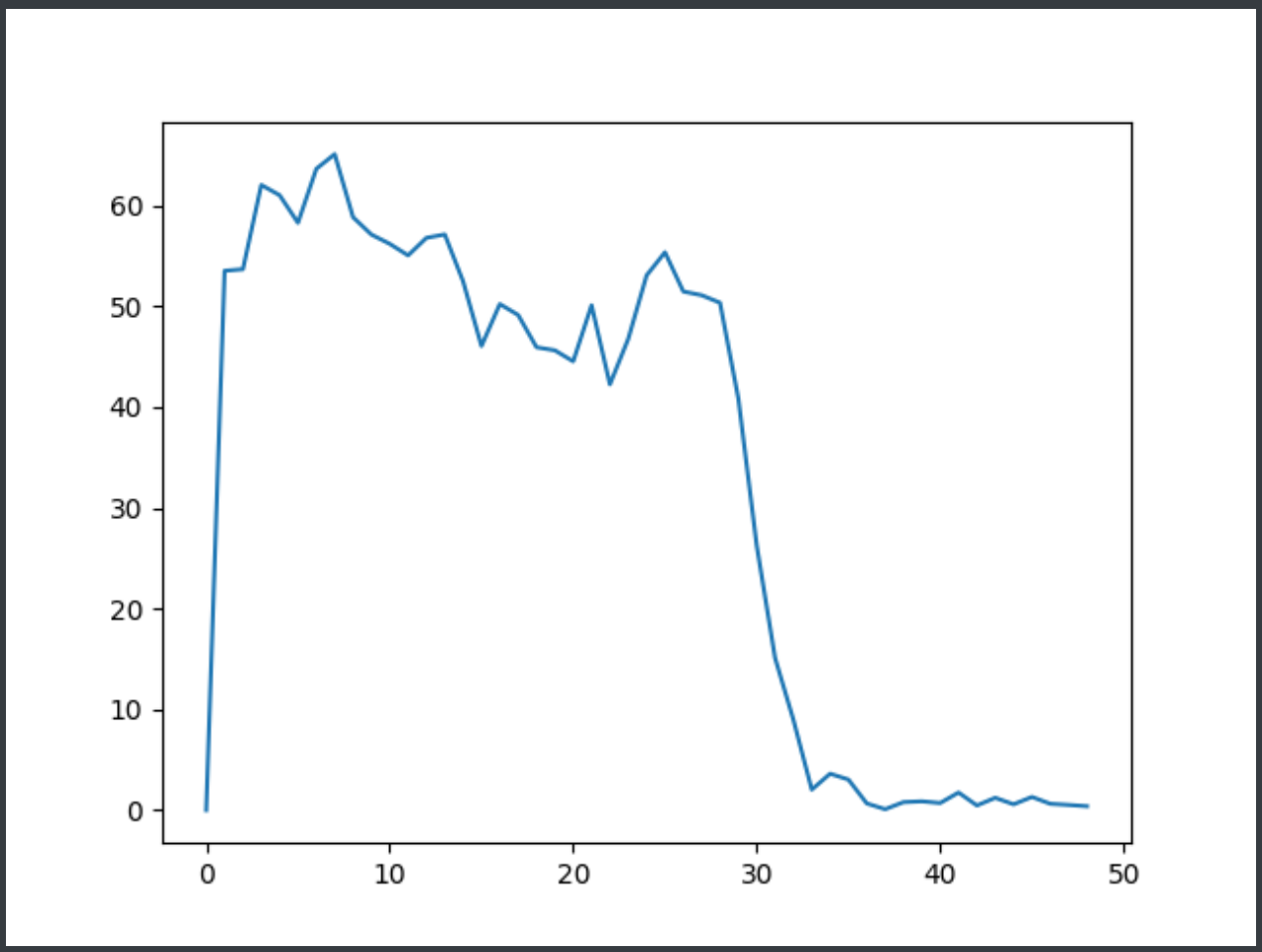
接着就是在当前的 16*16 方块附近找到 MSE 最小的 16*16 方块：

```
for hh in range(0, h-16):
  for ww in range(0, w-16):
    if hh >= x - radius and hh <= x + radius and ww >= y - radius and ww <= y + radius:
      subimage = transformer(new_data[hh:(hh+16),ww:(ww+16)])
      #mse = np.mean((subimage - target_block)**2) + np.mean((subimage - orig_target_block) ** 2)
      mse = np.mean((subimage - target_block)**2)
      if mse < min_mse:
        min_mse = mse
        min_mse_hh = hh
        min_mse_ww = ww
```

这里做过两种尝试：一种是直接和当前的块进行 MSE 的计算，一种是考虑了当前块和最初块的 MSE 求和。这里的 `transformer` 表示在求 MSE 前对块执行的预操作，如果是直接匹配像素，就是不变（identity）；如果是在 DCT 域上进行匹配，就是先运行一次 2D DCT，效果如下（修改代码中的 transformer 并运行 `python3 cars_pixel.py car`）：

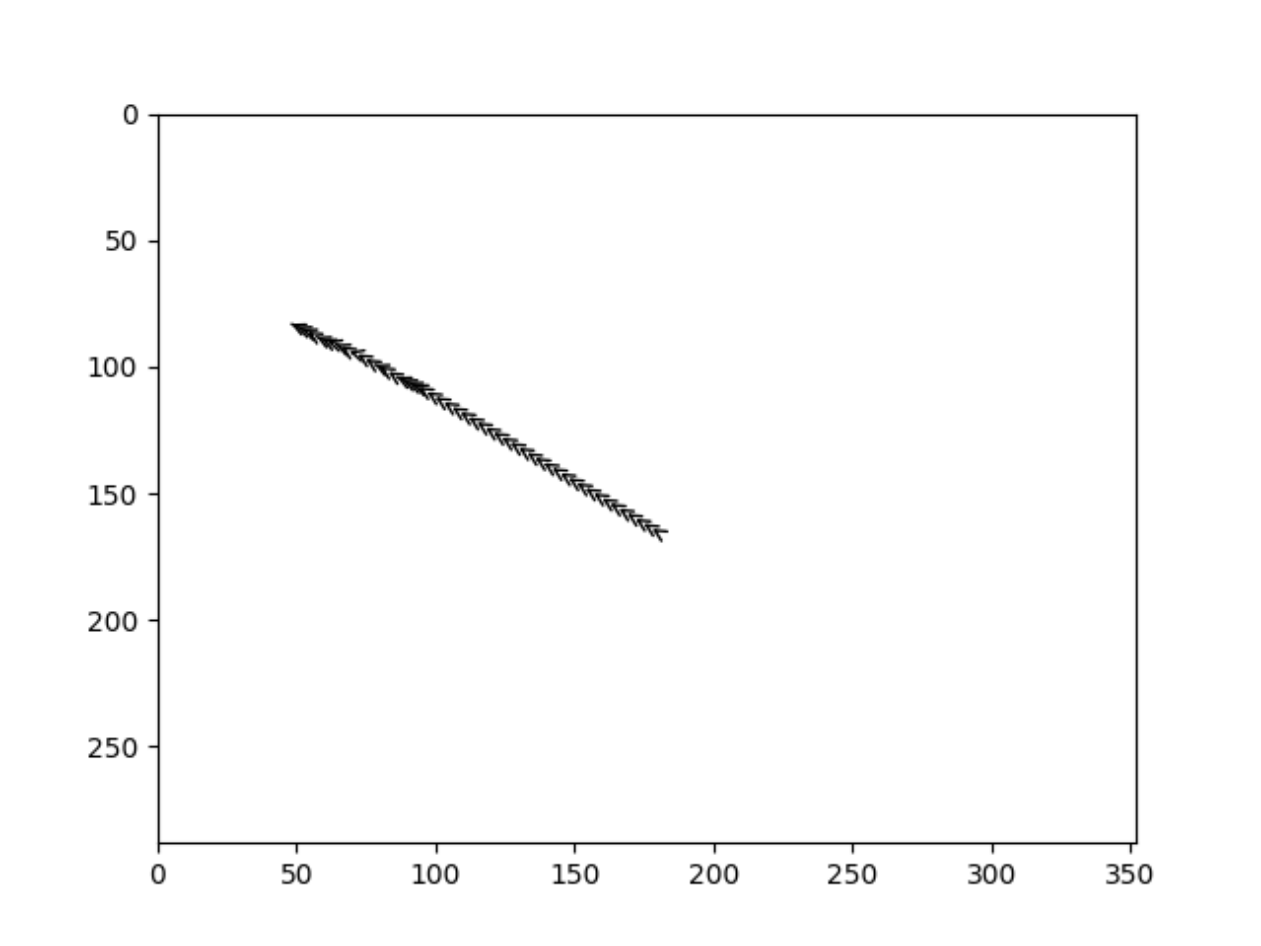
小车 MV 和 MSE：

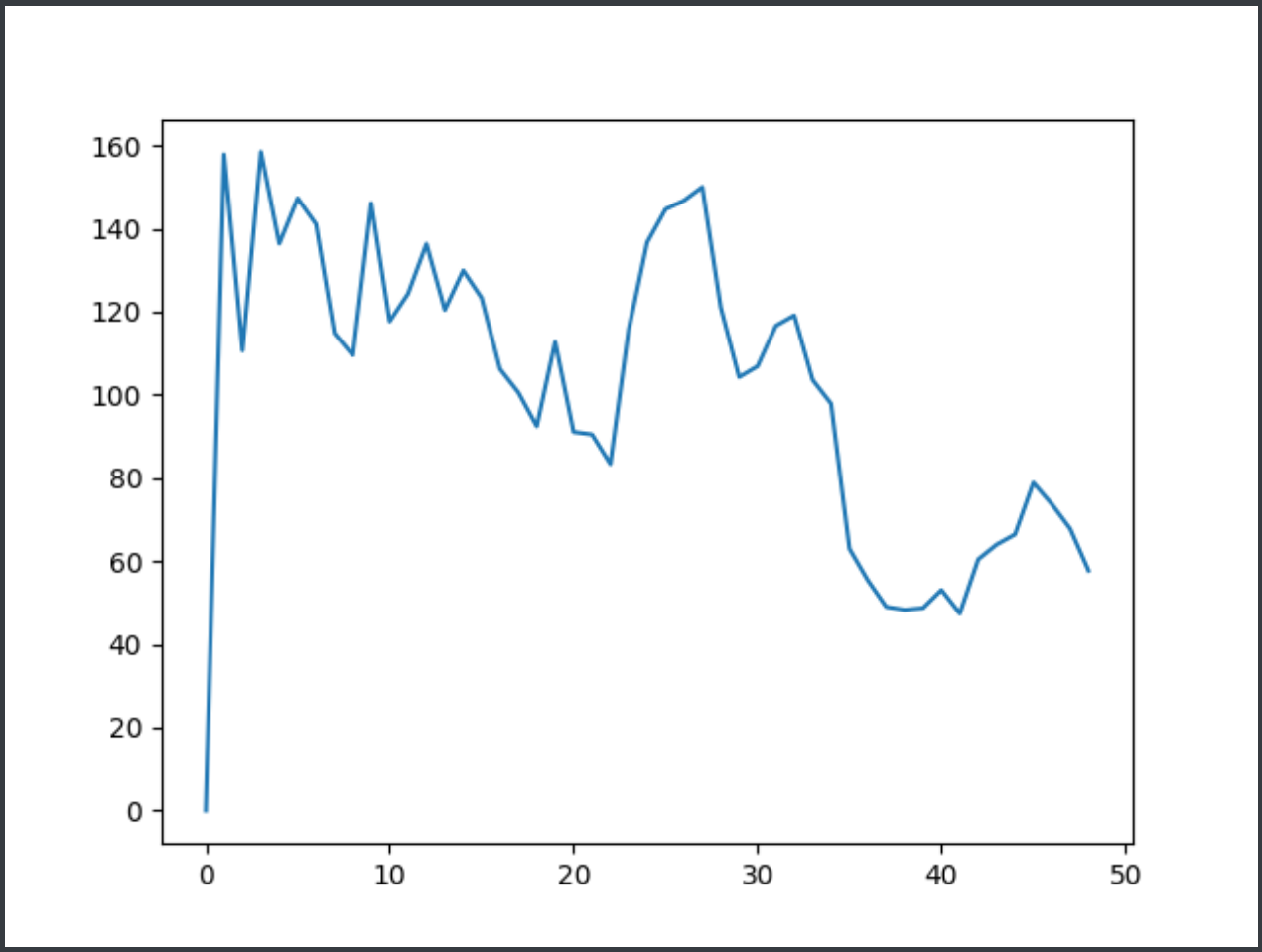

0
50
100
150
200
250
0
50
100
150
200
250
300
350


60
50
40
30
20
10
0
0
10
20
30
40
50

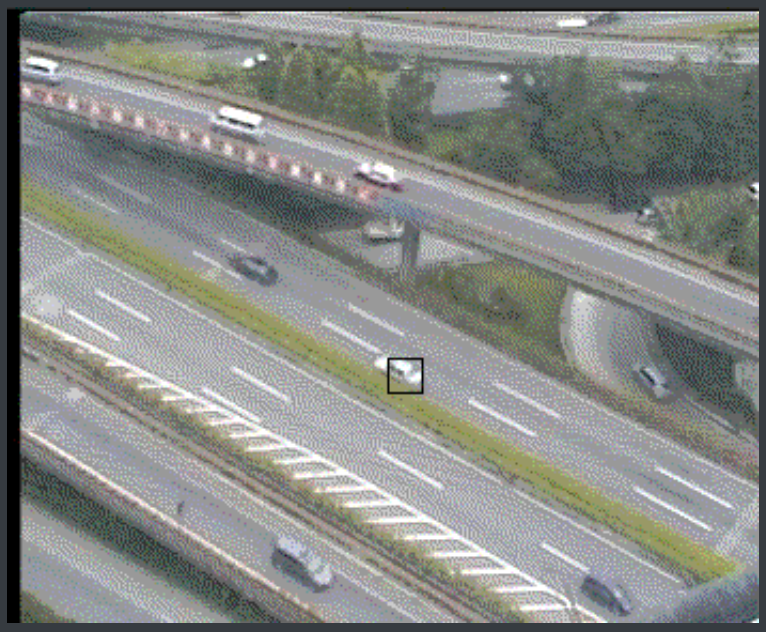
可以看到，在中间的时候，因为路上的白色和小车的白色一样，所以跟踪丢失了小车，到了附近的路上，之后就没有怎么变化了，所以 MSE 很小。如果按照上面所说的，把当前块和上一个块的MSE与当前块和初始块的MSE求和，就可以规避这个问题。

在 DCT 域上匹配效果：



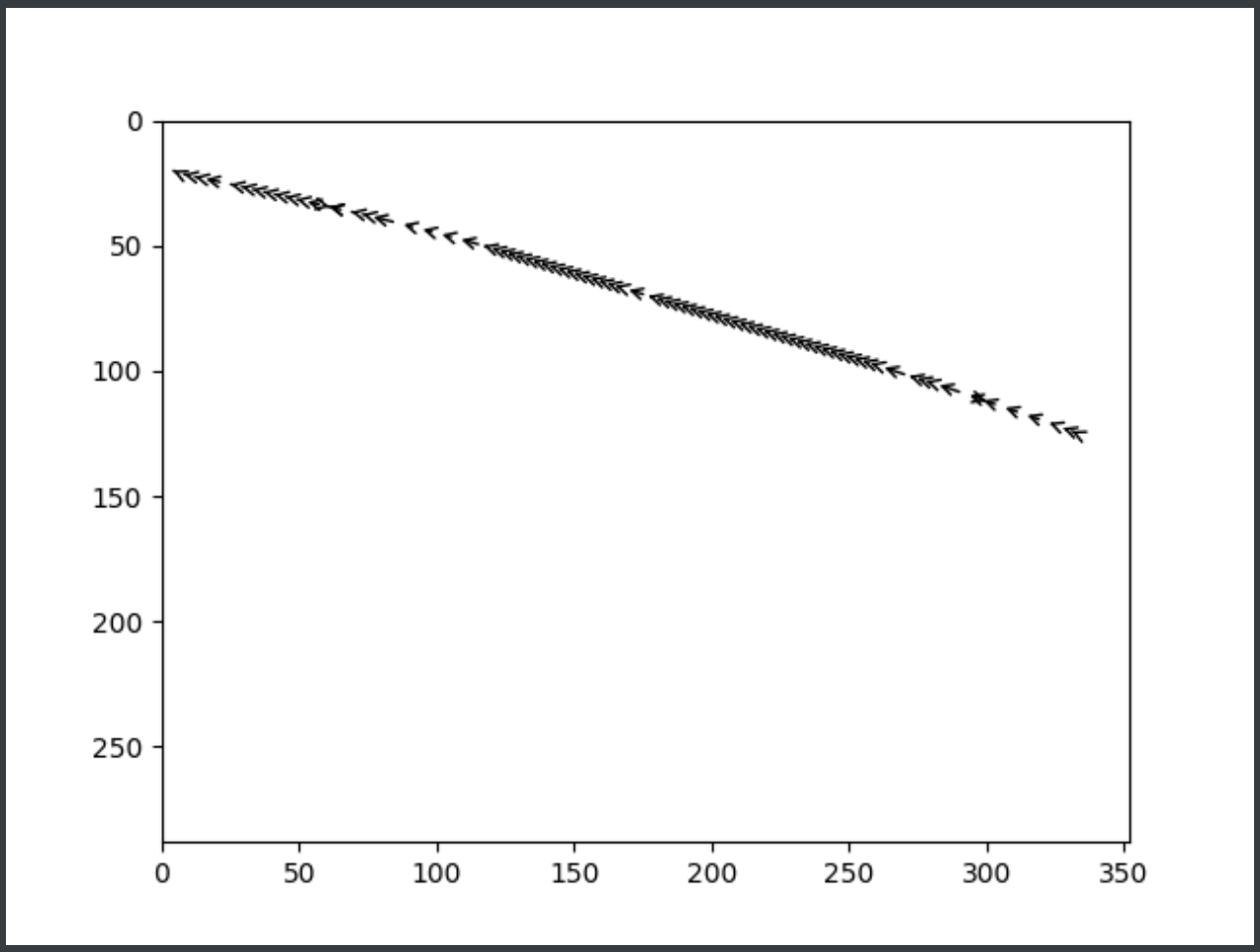


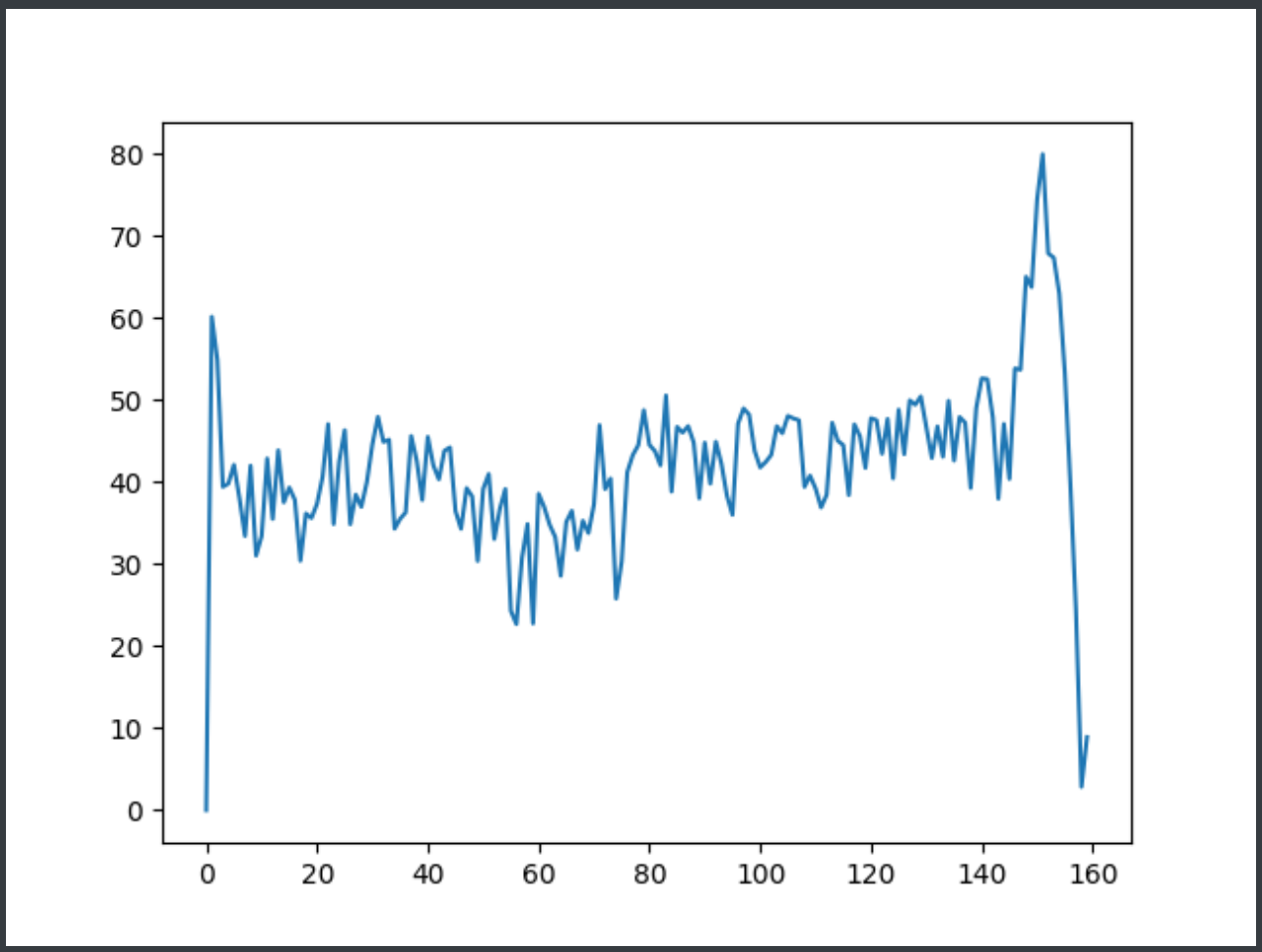
可以看到，在 DCT 匹配的时候，虽然路上也有白色，但是 DCT 的匹配能够抵抗住这种干扰，得到正确的跟踪效果（下面是个动图，路径为 car_detect/track_car_dct2d.gif）：



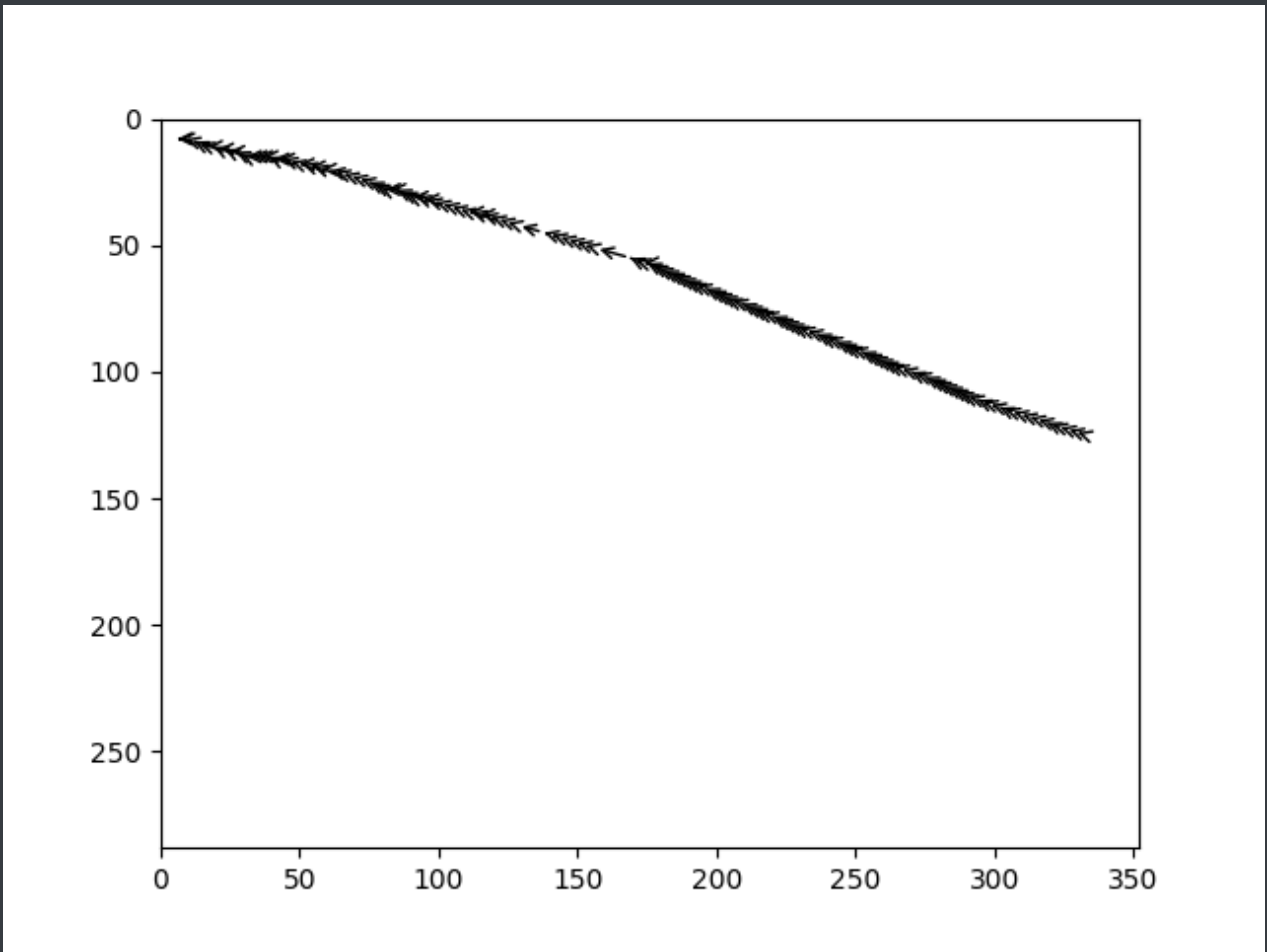
接着，对大巴进行类似的实验，效果比小车更加好，因为少了一些干扰：

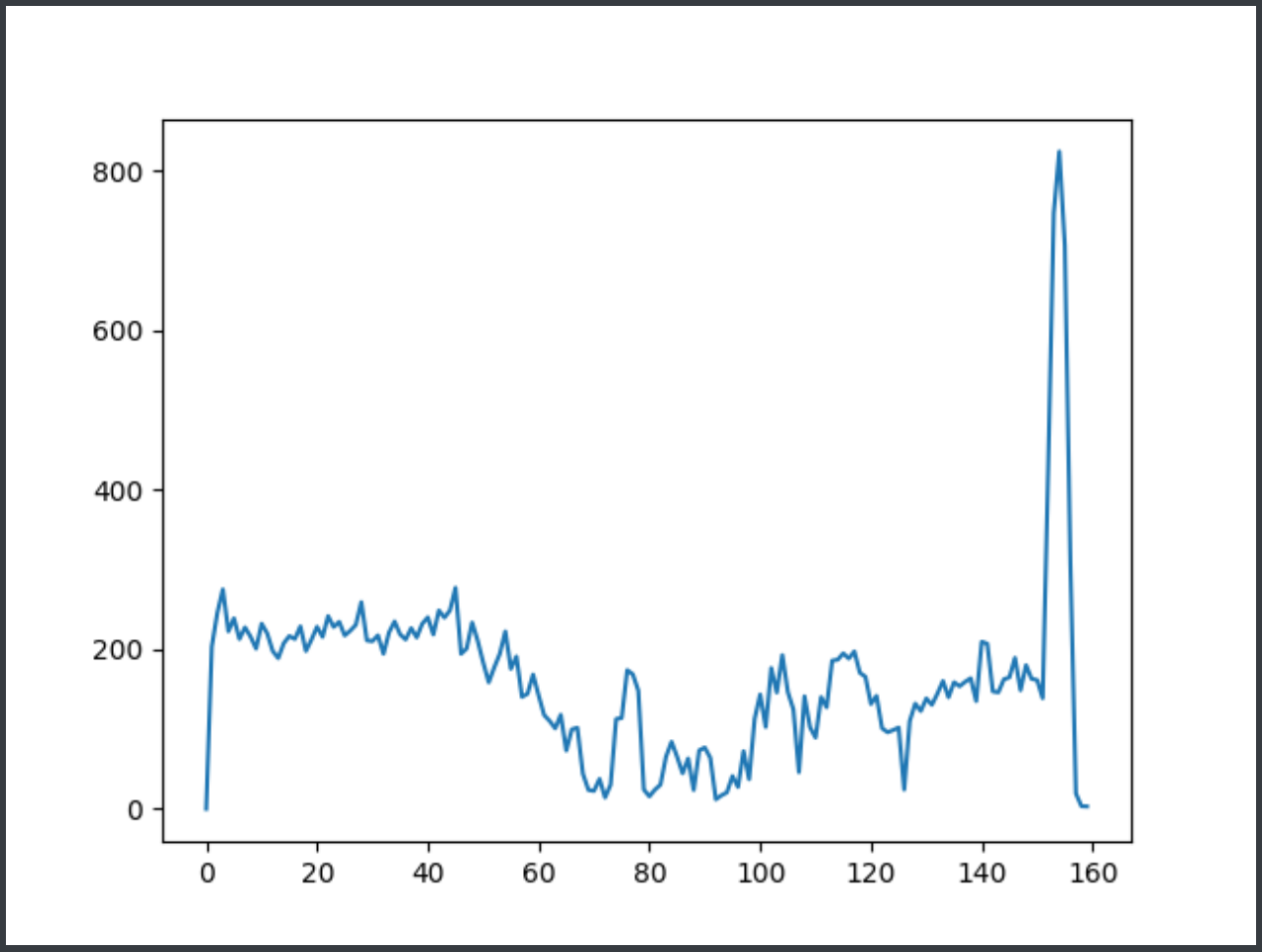
像素匹配：


0
50
100
150
200
250
0
50
100
150
200
250
300
350


80
70
60
50
40
30
20
10
0
0
20
40
60
80
100
120
140
160

DCT 匹配：





可以看到，最后大巴离开视频范围内的时候，MSE有一个明显的下降，也是符合预期的。因为大巴外形上的重复性，直接用像素匹配的话可以看到跟踪的时候会前后移动（car_detect/track_bus_identity.gif）：

相比之下 DCT 上的匹配时这个问题没有这么显著，但是也有新的“漂移”的问题（car_detect/track_bus_dct2d.gif）：

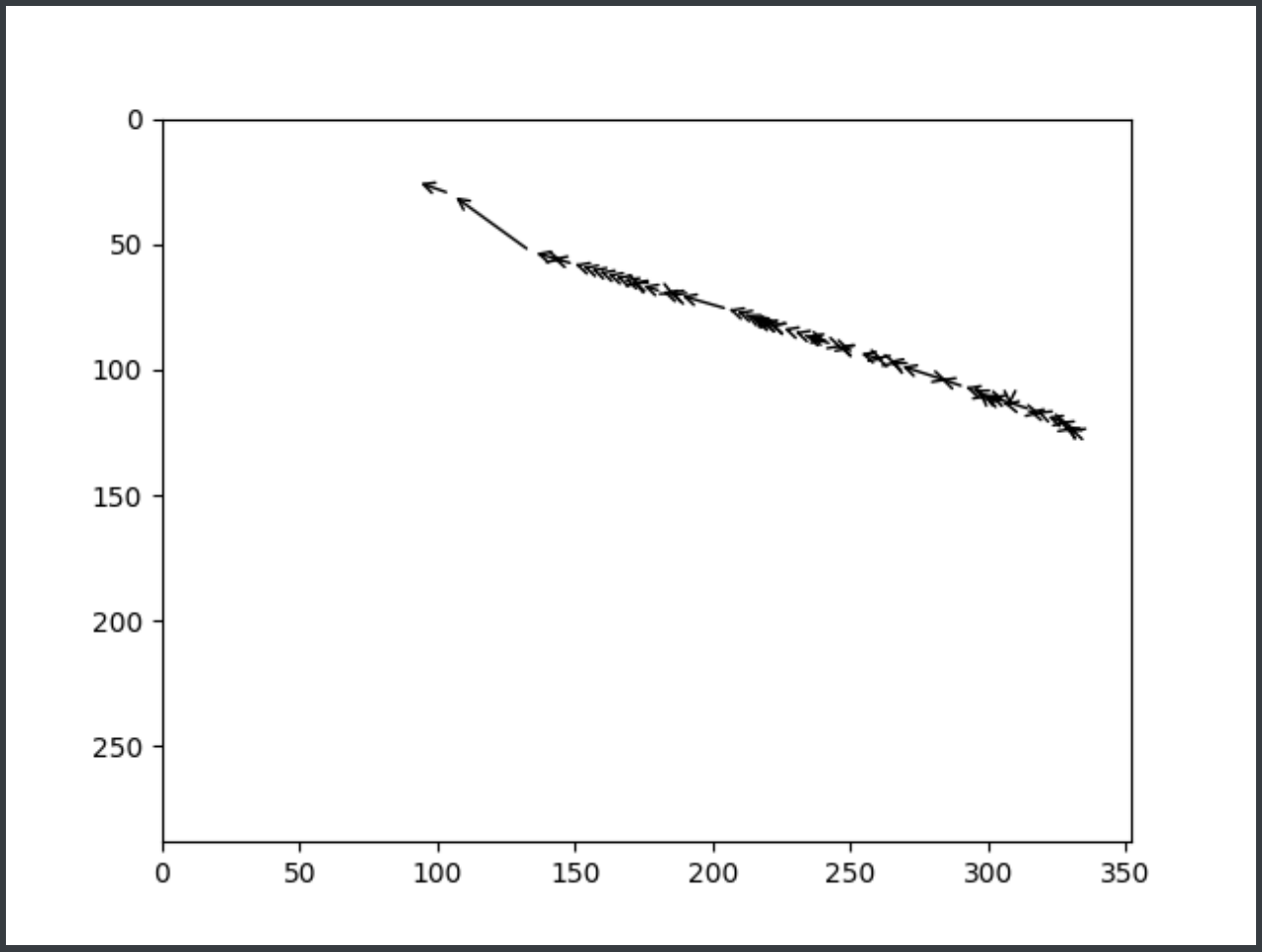
从性能上来看，当前的做法是对邻近的16*16的方块都进行 MSE 的计算，DCT则是分别计算2D DCT后计算 MSE，速度自然是前者比后者要快，毕竟少了一部计算。但是，实际在获取到图片的时候，可能能很容易得到图片中每一块的DCT（比如图片里可能就是存着每一块的DCT矩阵，显示的时候IDCT还原出来），如果可以复用这些 DCT 的结果，可能计算 DCT 的 MSE 会更好，但此时的 DCT 矩阵在图里的位置可能不会是连续的，这个时候移动跟踪的粒度就会比较粗（在上面的代码中，是逐个像素偏移来计算的）。

子任务 2 选取部分系数/像素进行运动估计

按照已有的代码框架，只要继续改 `transformer` 函数即可。试验了以下的函数：

```
def select(data):
    return data.digonal()
```

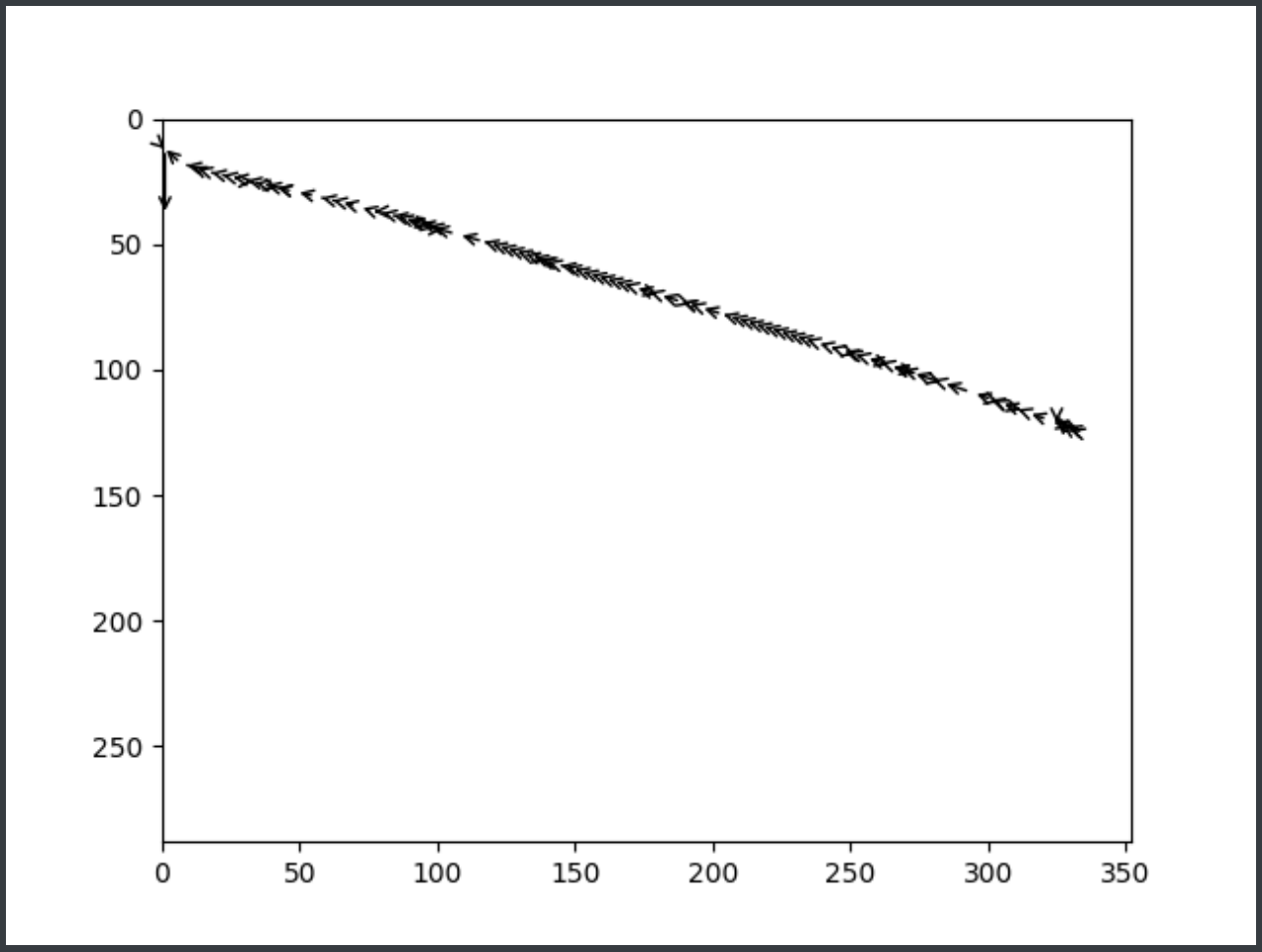
类似于 Exp1 中的思路，我只取一部分点，由于目前还是像素域的匹配，不能匹配过于集中，所以选择了主对角线上的点，效果如图：



可以看到，匹配到了比较长的距离，但是在中途还是跟丢了，这就是性能和效果之间的取舍问题。如果把两条对角线都跟踪上：

```
def select2(data):
    return np.concatenate((data.diagonal(), data.T.diagonal().T))
```

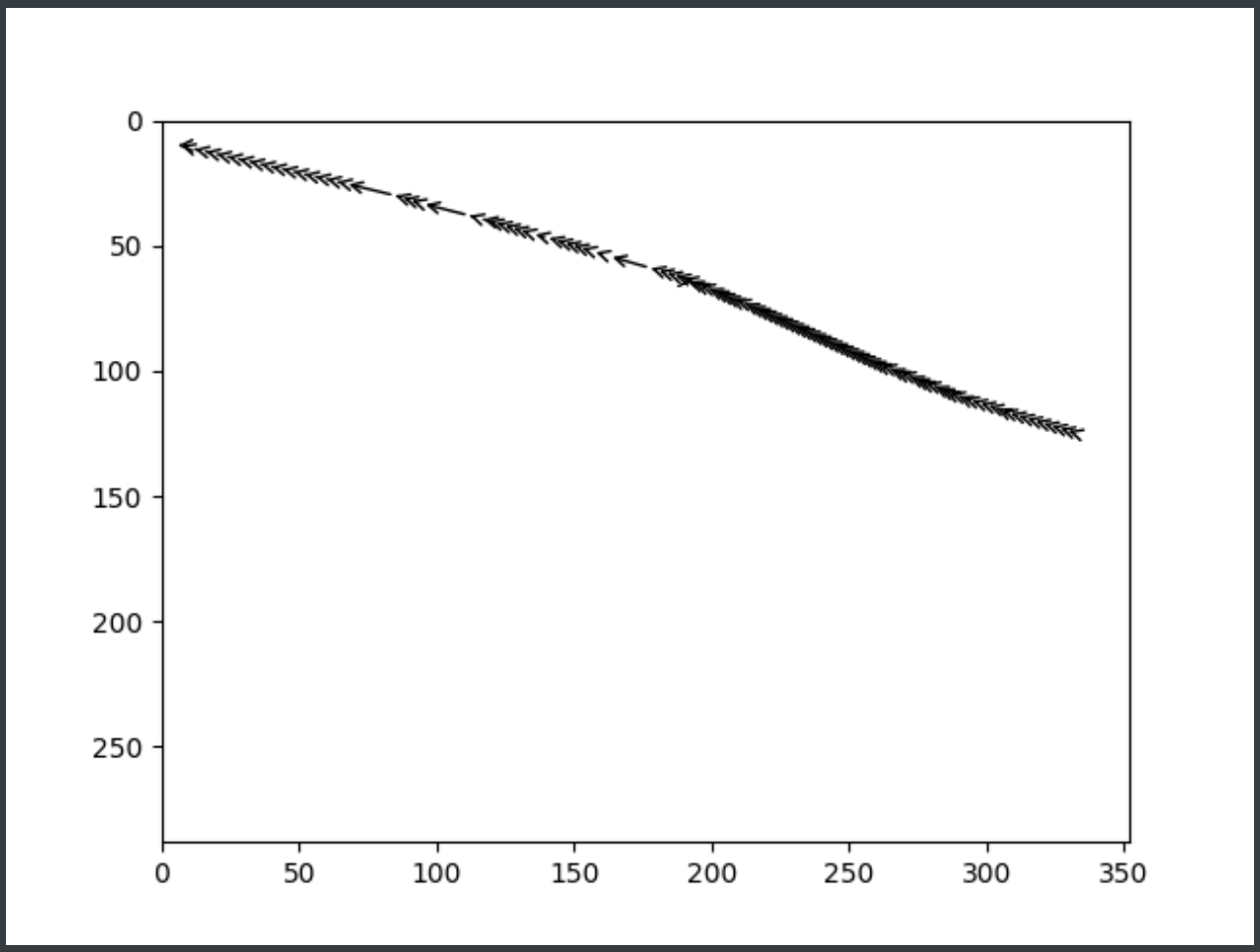
可以看到就没有出现刚才的问题：



如果在 DCT 上也做类似的事情，这次是保留左上角的分量：

```
1  def dct2d_select(data):
2      return dct2d(data)[0:8, 0:8]
```

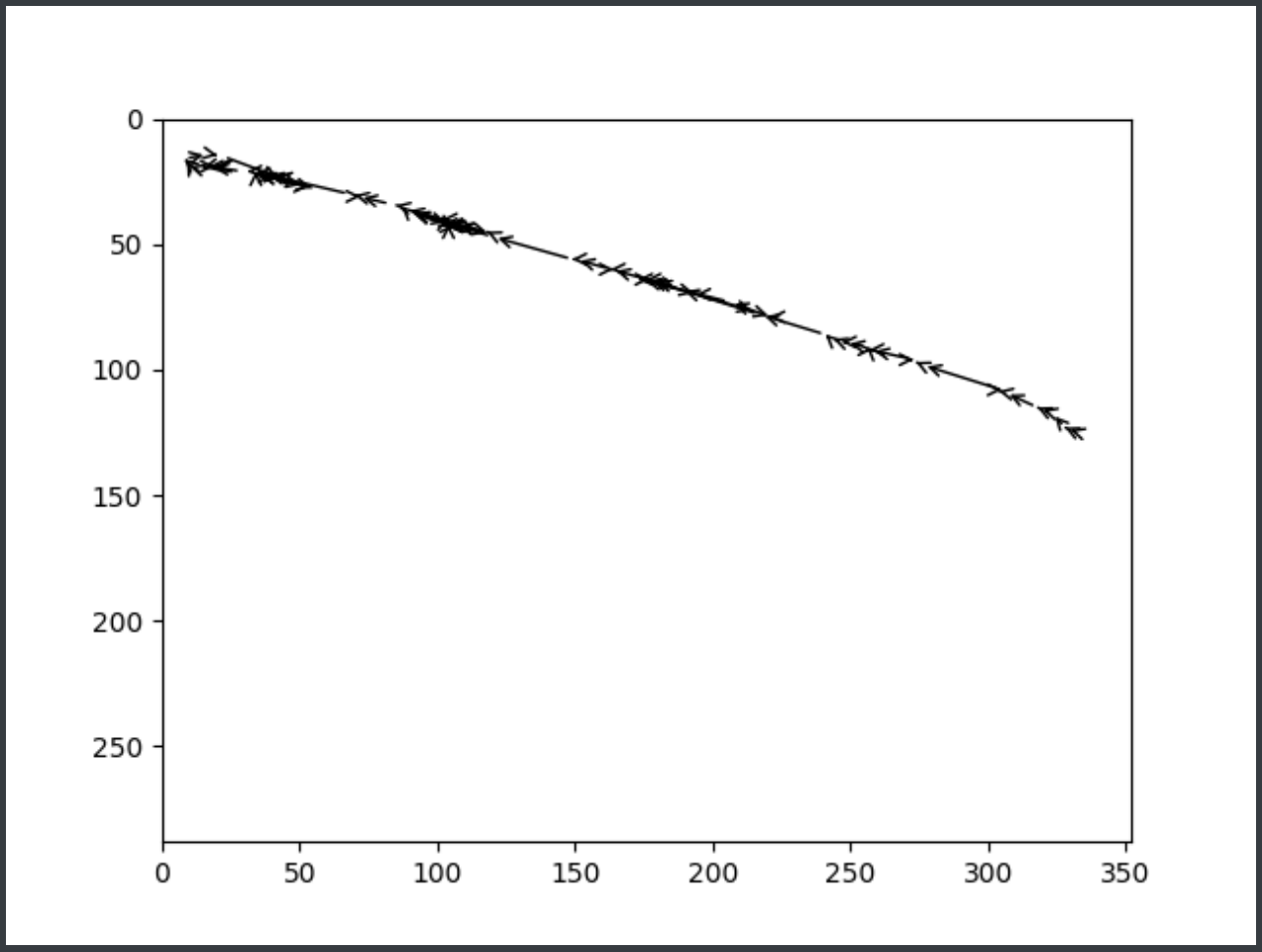
也得到了很好的跟踪效果：



如果继续减少使用的系数：

```
def dct2d_select2(data):
    return dct2d(data)[0:2, 0:2]
```

这个时候就会看到很大的不稳定性：

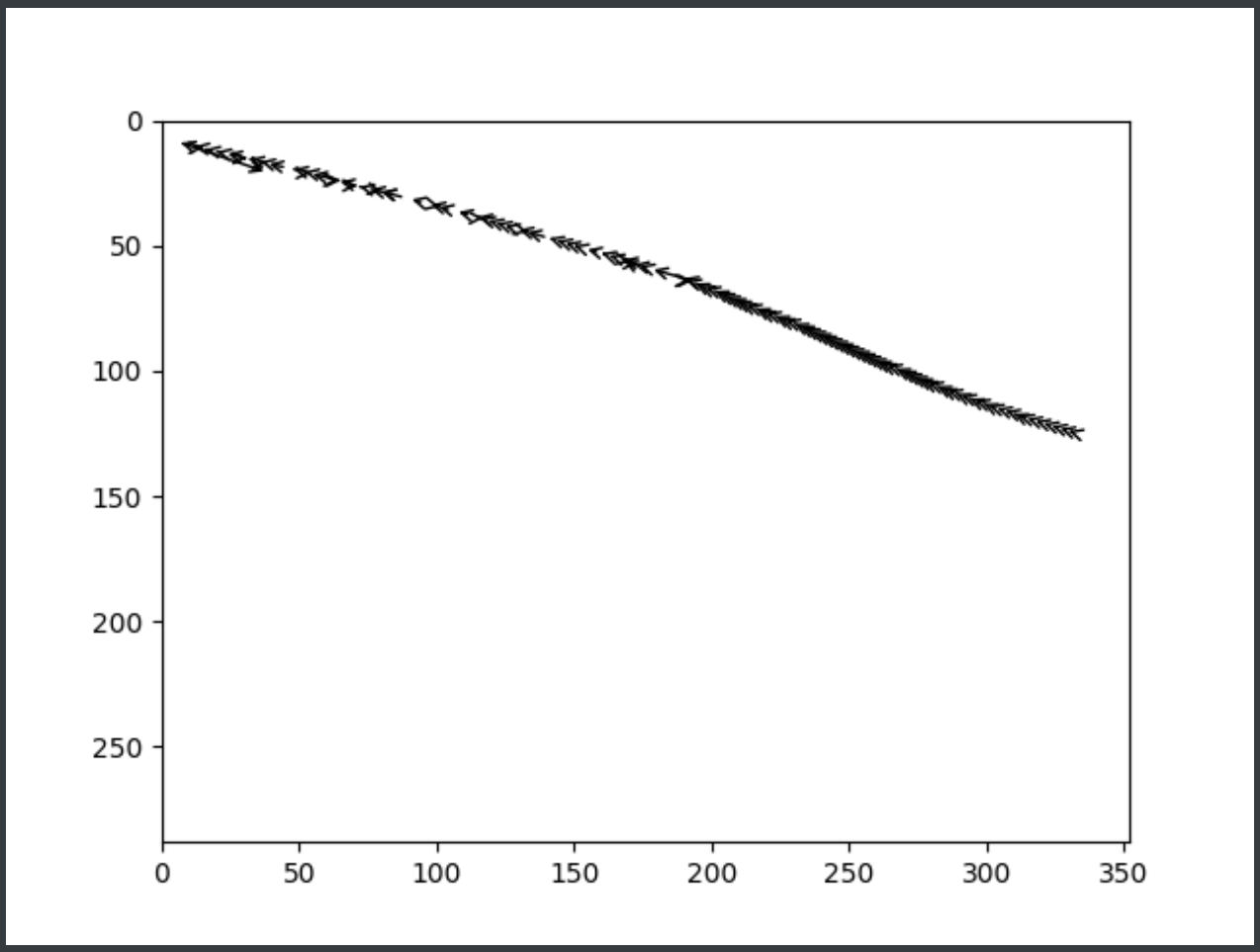


虽然路径是对的，但是中间其实很多时候跟踪到路上去了，虽然取了 DCT 中高频的分量，但还是丢失了很多细节。

接下来我又继续尝试了提取边缘特征，采用的是 Harris Corner Detection 算法，对应的函数：

```
def corner(data):
    gray = cv2.cvtColor(data, cv2.COLOR_BGR2GRAY)
    return cv2.cornerHarris(gray, 2, 3, 0.04)
```

也得到了比较不错的效果：



并且从动态图中可以看到确实一直在跟踪巴士的上边沿：

当然了，这一步额外的边缘计算，需要额外的时间，对于一些目标来说是比较合适的，例如这里的巴士，但不知道对于更多复杂的图形来说会不会也有比较好的效果。