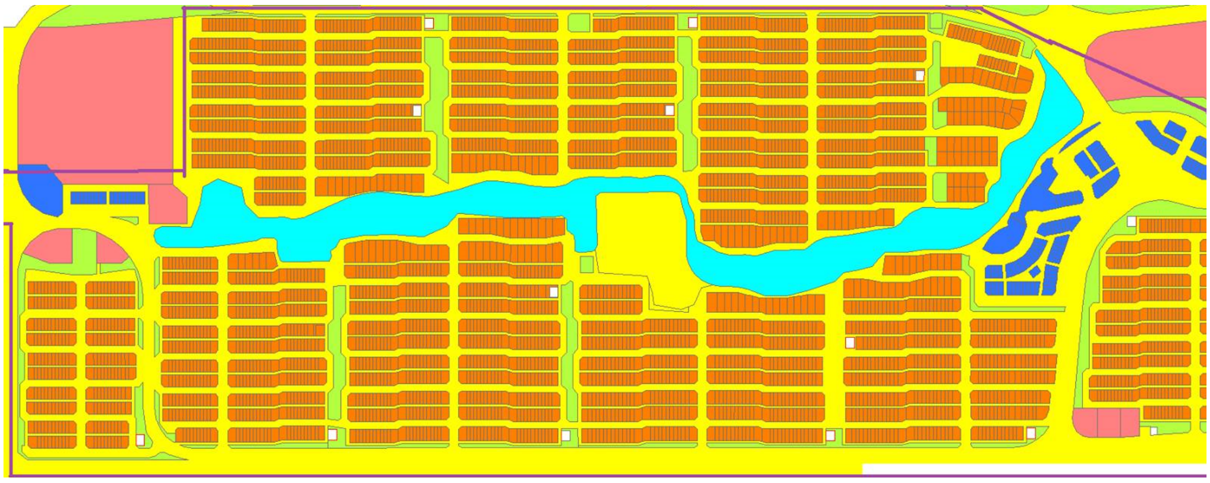
Initialization code

Example

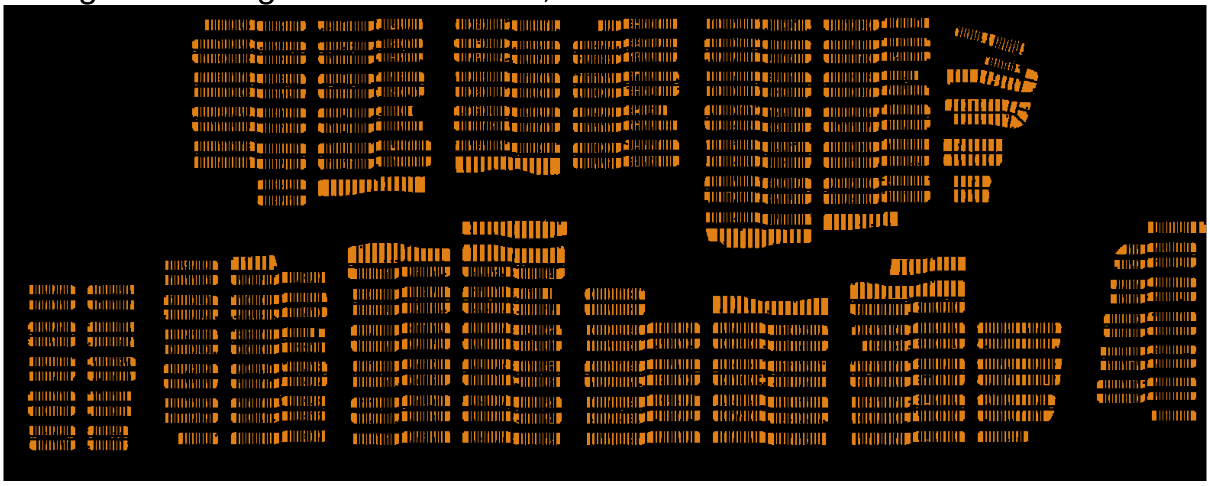
First, load the image file with all the orange boxes outside the boundary cropped out - the boundary line itself is not important.

```
SetDirectory[NotebookDirectory[]];
rawimg = Import["maps-2.png"];
rawimg // ImageDimensions
img = rawimg
```

```
{1755, 622}
```



We want to count the number of orange boxes in the image.
Hence, if we exclude all the other boxes (blue, green, pink, etc.), and then change the background to be black, then what we want should look like this:

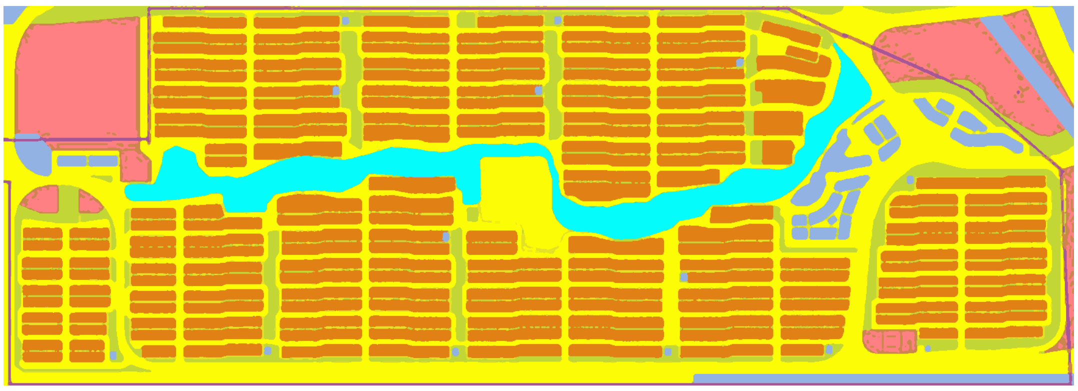


Call this desired image $\mathcal{I}_{\text{targ}}$. In this algorithm, we express this as a product of two images, that is, $\mathcal{I}_{\text{targ}} = f(\mathcal{I}_{\text{blob}}) \times \mathcal{I}_{\text{bound}}$, where:
→ $\mathcal{I}_{\text{blob}}$ is a coloured image (with a small number of colours allowed) that

depicts the general areas where each colour is located
→ f is a filter function that changes every colour except for the orange blobs to black - so $f(\mathcal{I}_{\text{blob}})$ shows the orange blobs on a black background
→ $\mathcal{I}_{\text{bound}}$ is the binary image that depicts the boundary for any coloured box
By multiplying $f(\mathcal{I}_{\text{blob}})$ with $\mathcal{I}_{\text{bound}}$, what effectively happens is that $\mathcal{I}_{\text{bound}}$ slices the orange blobs in $f(\mathcal{I}_{\text{blob}})$ to get the interiors of each orange box separated by a black background.

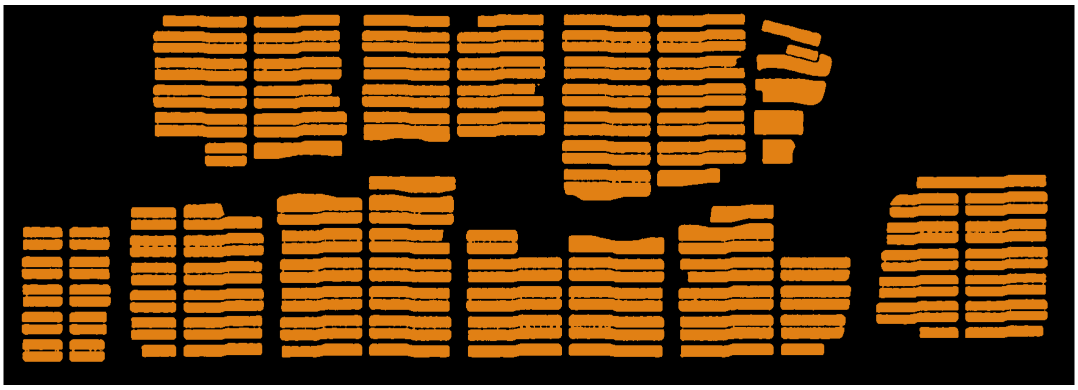
To get $\mathcal{I}_{\text{blob}}$, we need to use ColourQuantize - which takes the original image $\mathcal{I}_0$ and forces the image to use only some prescribed number of colours n_{colour} (we use $n_{\text{colour}} = 10$ in this case).
I usually use a GaussianFilter on $\mathcal{I}_0$ before applying ColourQuantize - it helps to blur the image slightly (makes it more blobby) - at worst it doesn't do much, at best it helps massively, so usually its a good idea to just use it.
After applying ColourQuantize, I usually apply a CommonestFilter, which helps to eliminiate artefacts of bits of different colours being in otherwise homogeneous blobs. Again, at worst it doesn't do much, but it does help a lot in some scenarios.
Parameters that can be manipulated at this point
→ n_{colour} - just choose the lowest number that gives reasonable looking blobs.
→ Dithering - true or false? I usually go with false, but sometimes true helps.
→ Integer parameter to CommonestFilter - usually the value 2 suffices, but if there are too many artefacts inside blobs then you might want to try increasing this.

```
qtz = ColorQuantize[GaussianFilter[img, 2], 10, Dithering → False];
qtzf = CommonestFilter[%, 2]
```



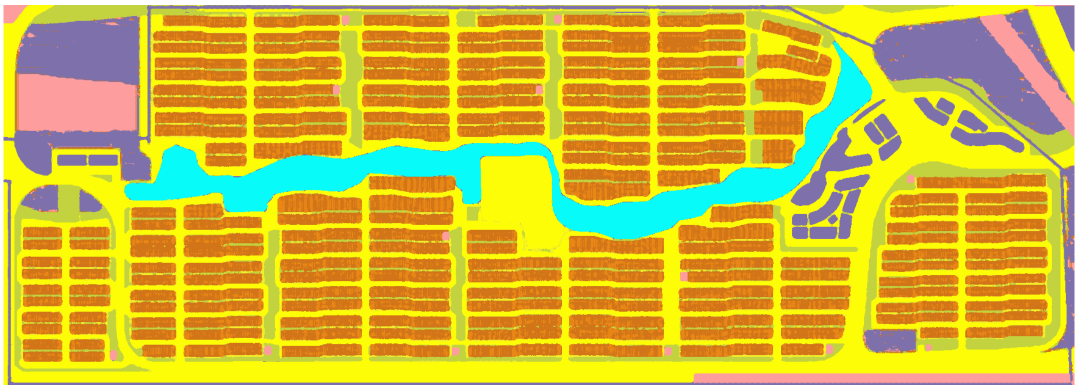
Next, apply the filter function f to $\mathcal{I}_{\text{blob}}$ by replacing every colour except orange with black - this should just leaves blobs of orange, giving $f(\mathcal{I}_{\text{blob}})$.
Not really any parameters to manipulate here.
So ideally $f(\mathcal{I}_{\text{blob}})$ looks like this:

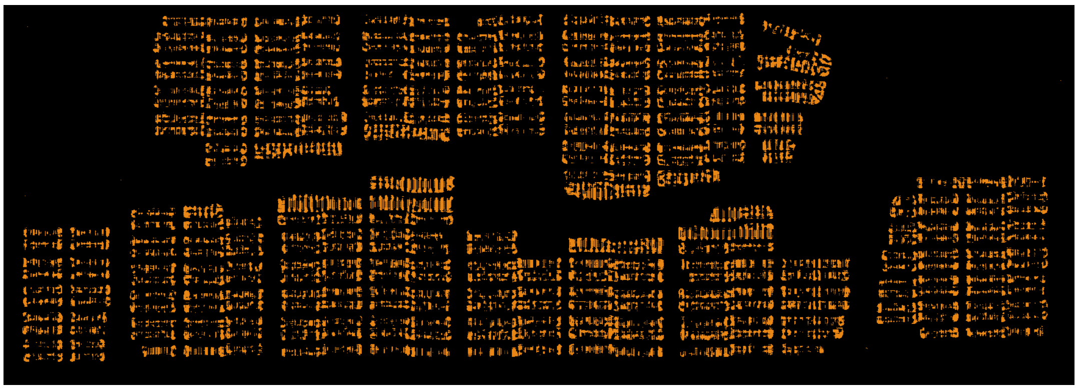
```
colours = DeleteDuplicates[Flatten[getrgb[qtzf], 1]];
ignorecls = {pickorange[colours]};
replc = {0, 0, 0};
imgflt = filtercolours[qtzf, ignorecls, replc]
```



Poor choices of parameters, however, can give a bad result, like as follows (removed Gaussian filter, lowered n_{colour} to 8, and enabled dithering):

```
qtz2 = ColorQuantize[img, 8, Dithering → True];
qtzf2 = CommonestFilter[%, 2]
colours2 = DeleteDuplicates[Flatten[getrgb[qtzf2], 1]];
ignorecls = {pickorange[colours2]};
replc = {0, 0, 0};
filtercolours[qtzf2, ignorecls, replc]
```

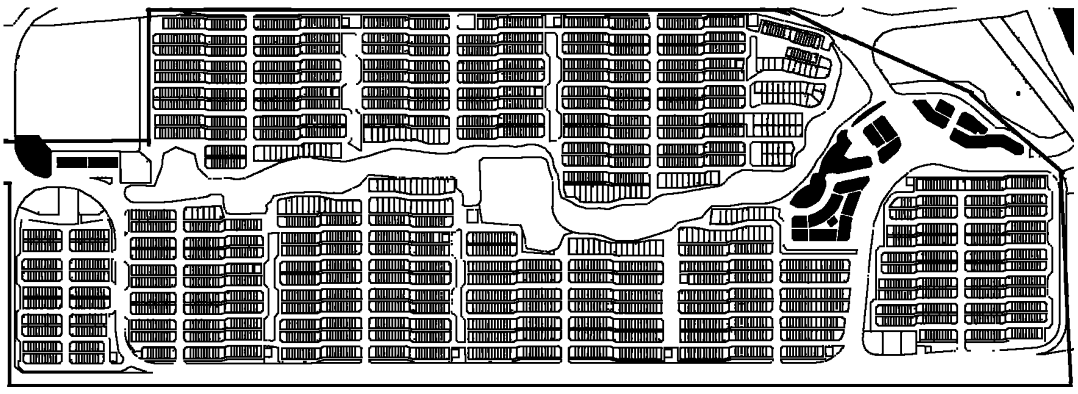




Next, we consider the separate problem of obtaining $\mathcal{I}_{\text{bound}}$. The key function here is MorphologicalBinarize (its essentially magic) - in all cases examined

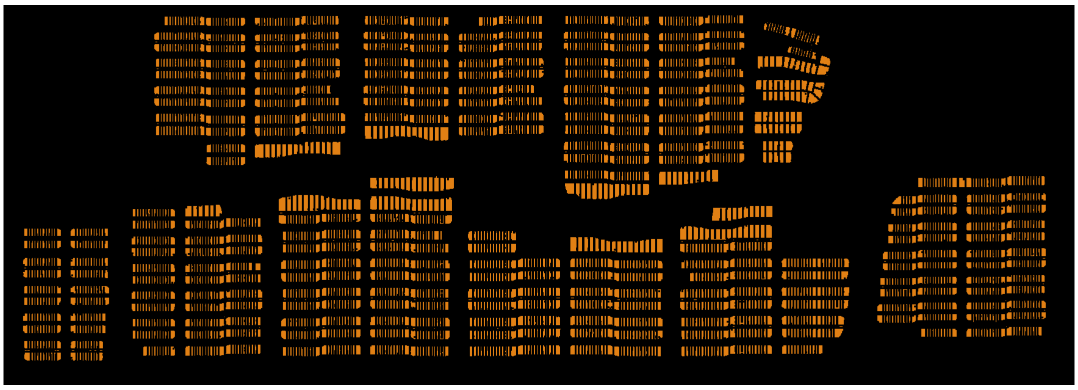
so far, it has (with some help needed sometimes) sucessfully provided the boundary of all coloured regions. Note that $\mathcal{I}_{\text{bound}}$ is not required to distinguish between an orange box or a blue box (which because reasons gets converted to black) or a green box or a pink box - it just needs to have at least the boundary for the orange boxes, but it can show the boundary for other coloured boxes as well.
There are three important parameters:
→ Real parameter *r* to the MorphologicalBinarize function - must be a real value in the interval [0, 1] - chosen by trial and error. If *r* is too small, then most boundaries will not register or will be very fragemented (we require it to be continuous). If *r* is too large, some of the interior of (non-blue) cells will suddenly be filled with black, so you end up losing valid cells by virtue of it being filled. Usually $r \in [0.55, 0.75]$.
→ Applying Sharpen to $\mathcal{I}_0$ before using MorphologicalBinarize - in some cases (like this one) it is required to produce reasonable boundaries, in most other cases don't use it if you don't need to.
→ Using a MinFilter after the MorphologicalBinarize function - Use this if (like in this case) the boundaries from the procedure is too thin - otherwise, just don't.

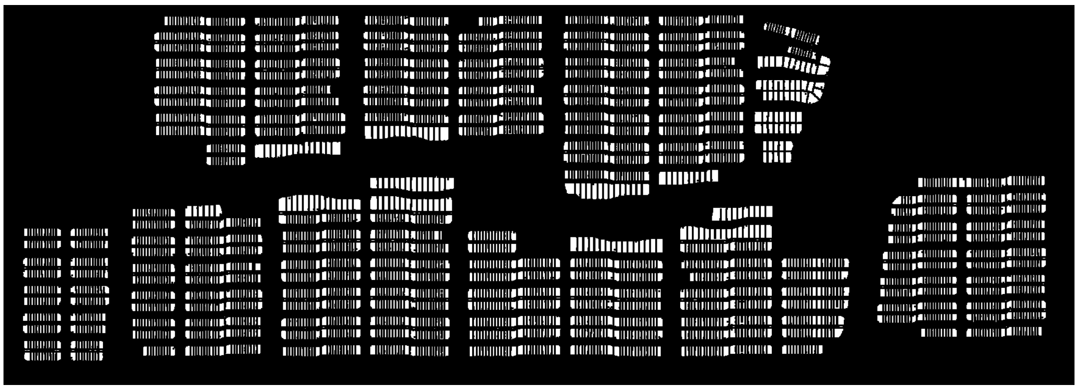
```
imgbound = MinFilter[MorphologicalBinarize[Sharpen@img, 0.61], 1]
```



Then, take the product with $f(\mathcal{I}_{\text{blob}})$ in order to get an image with orange blobs subdivided according to the boundaries in $\mathcal{I}_{\text{bound}}$, giving us $\mathcal{I}_{\text{targ}}$ as below. Binarize this image in preparation for the last step.

```
ImageMultiply[imgflt, imgbound]
tc = Binarize@%
```





Finally, we have white blobs (each representing an orange lot) that are separated by a black background.
To count the number of white blobs, use the countblocks function, which takes in two parameters:
→ First parameter: The binarized version of $\mathcal{I}_{\text{targ}}$
→ Second parameter: A positive integer ζ that treats blobs of size ζ (that is, blobs consisting of ζ pixels) and smaller as not contributing to the count. This parameter is exceedingly useful in getting rid of artefacts such as stray white points or small blobs that aren't derived from an orange lot.
The countblocks function gives the return value $\{r_1, r_2\}$, where r_1 is the list of blob sizes of size greater than ζ (so the length of r_1 is the count of blobs that we want) and r_2 is the image formed by these blobs of size greater than ζ only. This second parameter is useful as a comparison to the binarized version of $\mathcal{I}_{\text{targ}}$, since the missing blobs indicate which blobs weren't counted. To faciliate this comparison, we can use ImageDifference, as below.

The ImageDifference here shows some issues with classifying the blobs in the upper right quadrant as part of the count - this probably stems from the boundaries of that section being too thick (however, removing the MinFilter to fix this would probably affect other parts of the image, which require the MinFilter to get a reasonably sized boundary). Nonetheless, the ImageDifference can show which sections aren't being counted well, and one can do a manual count of those sections instead and add them onto the count if desired.

```
{cts, resimg} = countblocks[tc, 8];
cts // Length
ImageDifference[tc, resimg]
```

```
4045
```

To check the results of the countblocks function, we can isolate a small part of the image (here the rows 1-84, columns 240-400) and then compare the results of the countblocks function (it counts 81, while getting rid of artefacts) with a manual count of the original image (also 81).

```
temp = ImageTake[tc, {1, 84}, {240, 400}]
{cts, resimg} = countblocks[temp, 8];
cts // Length
GraphicsRow[{ImageTake[rawimg, {1, 84}, {240, 400}], temp, resimg}]
ImageDifference[temp, resimg]
```

```
81
```