```
        -------
        (N,2) numpy array
            The corresponding (lon, lat) coordinates
        """
        if type(pixel) != np.ndarray:
            pixel = np.array(pixel).reshape(1,2)
        assert pixel.shape[1]==2, "Need (N,2) input array"
        pixel = np.concatenate([pixel, np.ones((pixel.shape[0],1))], axis=1)
        lonlat = np.dot(self.M,pixel.T)

        return (lonlat[:2,:]/lonlat[2,:]).T

    def lonlat_to_pixel(self, lonlat):
        """
        Convert a set of lon-lat coordinates to pixel coordinates
        Parameters
        ----------
        lonlat : (N,2) numpy array or (x,y) tuple
            The (lon,lat) coordinates to be converted
        Returns
        -------
        (N,2) numpy array
            The corresponding (x, y) pixel coordinates
        """
        if type(lonlat) != np.ndarray:
            lonlat = np.array(lonlat).reshape(1,2)
        assert lonlat.shape[1]==2, "Need (N,2) input array"
        lonlat = np.concatenate([lonlat, np.ones((lonlat.shape[0],1))], axis=1)
        pixel = np.dot(self.invM,lonlat.T)

```

```
62          return (pixel[:2,:]/pixel[2,:]).T
```

pixel_mapper.py hosted with ♥ by GitHub view raw

calculate the speeds at which they are travelling.

The camera feed

For this article we'll be using a few minutes of video taken from the livestream below, provided courtesy of Provincie Gelderland (and streamed using VidGear).

You can see that the camera is fixed in position and observes a set of objects on an approximately 2D surface — vehicles travelling around a roundabout. We should therefore be able to define a matrix transformation to map the real space positions of these vehicles. But before we can perform this mapping, we will first need to detect and track the vehicles within the video. We'll do this with two out-of-the-box techniques: a pre-trained TensorFlow object detector and the SORT tracking algorithm.

Detecting

First we need to generate the detections. Fortunately, the category of object we wish to detect — "car" — is one of the 80 COCO object classes, so we can use a TensorFlow object detector pre-trained on the COCO dataset. For this article we'll use an Inception SSD (ssd_inception_v2_coco) and generate the detections by following the walkthrough provided by TensorFlow, keeping only those classified as "car".

Although it struggles with distant and occluded vehicles, this object detector does a pretty good job for a pre-trained model:

Tracking

Next we need to track these detections across the video. We'll do this by first dropping any "duplicate" detections (that have a non-negligible IoU with a more probable detection in the same frame) and then using the SORT algorithm to track the objects between frames. Finally we ignore any tracked objects that have fewer than five detections. To turn these tracked detections into paths we take the bottom center of the bounding box generated by the Kalman filter to be the vehicle's position.

This approach allows us to track the individual paths of all detected vehicles around the roundabout, with only some slight confusion between tracks for close-travelling vehicles in the far distance:

Mapping

Now that we've run our detection and tracking algorithms, we have, for each vehicle, a set of u, v coordinates which represent the path it took in "pixel space", i.e. the 2D frame which our camera is recording. To know the "real space" location of each vehicle, we need to map these coordinates to another (approximately) 2D plane — the Earth's surface. We can perform this mapping with a **perspective transform**, a matrix operation which

projects a set of points from one 2D plane to another.
Adapted from: Fundamentals of Texture Mapping and Image Warping, Paul Heckbert

The maths for deriving this transformation can be a little involved (if you like linear algebra, they can be found here), but fortunately the matrix calculation can be performed by the OpenCV function getPerspectiveTransform. All we have to do is provide the coordinates of four points — noncollinear, so defining the corners of a quadrilateral — in both coordinate systems.

This is the fiddly part. To get these four coordinates, we need to identify the longitude-latitude of four pixel locations in our frame. These should be spread as far apart as possible: the greater the area covered by the quadrilateral, the more accurate the perspective transformation will be. For our video, I have identified four clear landmarks in the frame and located these on a satellite image (via google maps). These are defined and plotted below.

We then define a "*PixelMapper*" class which, from these four points, can translate a *(u,v)* pixel location to a *(lon,lat)* real world location, and vice versa.

Note that *PixelMapper* simply performs a linear transformation of coordinates: there is no logic to determine whether a background pixel falls beyond the vanishing point of the 2D surface, or whether a longitude-latitude coordinate falls behind the field of view of the camera. We therefore need to be careful of these cases when making transformations.

We can then perform these transformations like so:

And applying this transformation to the *(u,v)* coordinates of our tracked objects, we can now plot their real world positions as they travel around the roundabout:

Speed calculation

As our final step we want to calculate the speeds of the vehicles we have tracked. To do this we need to calculate the distance they travel, which we can do using the longitude-latitudes we've just calculated. Rather than doing this directly (using the haversine formula) we will instead perform another coordinate transform, this time converting the vehicle latitude-longitudes into a local Cartesian coordinate system, with units in metres. This then allows us to easily calculate lengths and areas in these units.

We will transform our tracks from longitude-latitude (epsg:4326) to RD

New (epsg:28992), a Cartesian coordinate system (with units in metres) that is valid across the Netherlands. (Local coordinate systems for any location can be easily searched for here). This can easily be done using pyproj:

Then for our final piece of maths: to calculate the vehicles' speeds we need only to divide the distance — calculated from the coordinates above — by the time. In the animation below we're doing this using the distance travelled across the previous 10 frames (corresponding to 2 seconds of time). Fortunately for this video it seems that all drivers are well below the speed limit.

About HAL24K

HAL24K is a Data Intelligence scale-up based in San Francisco, Amsterdam and Singapore, delivering operational and predictive intelligence to cities, countries and companies.

The Science of Wait Time: How We Built Facility Insights

Piero Cinquegrana in keeptruckin-eng

FireMind: Using Neural Networks to Model Wildfire

Michael Burnam-Fink in MBF-data-science

Whether the country is still the decisive factor in one’s income?

Patrick So

An Introduction to the Standard Data Tabulation Model (SDTM)

DEVENDER PALSA

The Monty Hall Problem

Jeffrey Hanif Watson in Towards Data Science

Model Selection & Assessment

Michał Oleszak in Towards Data Science