Advanced Lane Finding

December 8, 2017

The goal of this project was to identify curved lane markings from hood- or dash-mounted camera view. Broadly, the complete pipeline used three steps: 1. Correct for barrel distortion and use a four-point perspective transformation to produce a top-down view. 2. Use a combination of thresholding, image derivatives, and other OpenCV effects to extract only the pixels associated with the two lane markings. 3. Produce polynomial fit lines to the found pixels, both in image space, and in world space. Use the latter to estimate the radius of curvature of the lane, and the car's deviation from center (in meters).

```
In [1]: import numpy as np, matplotlib.pyplot as plt, cv2
        import cvflow as cf
        import laneFindingPipeline
        from utils import ShowOpClip
        from jupyterTools import src, propertySrc
        %matplotlib inline
```

```
In [2]: allFrames = laneFindingPipeline.utils.loadFrames()
        showOpClip = ShowOpClip(allFrames['harder_challenge'][:32])
        frame = allFrames['project'][-1]
```

1 Preprocessing pipeline

1.1 `cvflow` for visualizing OpenCV pipelines

I fell down a rabbithole on this project implementing a computational graph library for assembling my OpenCV pipelines. This was inspired (obviously) by TensorFlow. Though here I use such an approach merely for convenience of introspection rather than for allowing for code generation, I could see that being an alternate use. Towards the end of this report, I'll list this and a few other possible future improvements to this library.

The basic object in cvflow is the `Op` (aka "node"), which wraps some operation, such as a simple `Add` of two or more arrays, or more relevantly, something like `cv2.cvtColor`. Crucially, `Op` objects contain a `parents` list, which they use directly in their wrapped operation, a `object.value` property which returns the result of this operation, and a `children` list, each of whom have this object as a parent and make use of its `object.value` property.

These nodes form a directed acyclic graph (DAG) (though I do no explict test for acyclicity, I rely on it throughout to avoid `RecursionErrors`), so the `object.value` properties are cached on their first evaluation and not recomputed for use by multiple children. A special `Op` subclass called `BaseImage` (with subclasses `ColorImage` and `MonoImage` acting as further offenses in the ongoing

violation of YAGNI that is this library) calls `self.invalidateCache` on the setter of the `self.value` property, which recursively calls itself on the children of the `BaseImage`.

```
In [3]: op = cf.Op()
        src(op.invalidateCache)
```

```
Out[3]:
```

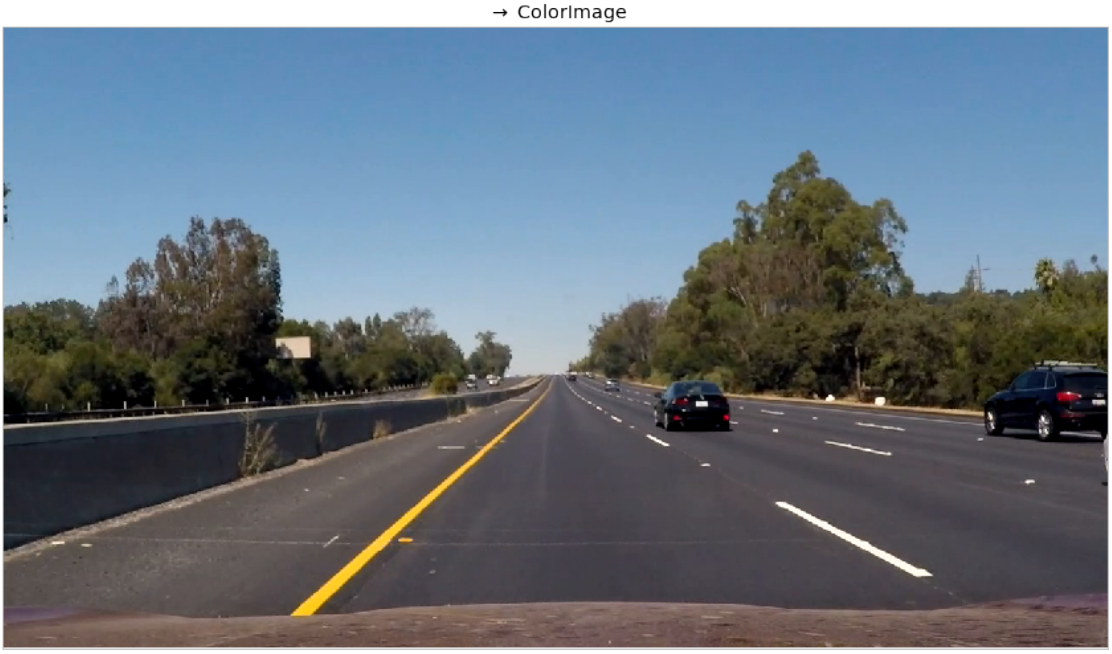
Lines 90 throuh 93 of op.py:

```
 def invalidateCache(self):
     misc.clearCache(self)
     for child in self.children:
         child.invalidateCache()
```

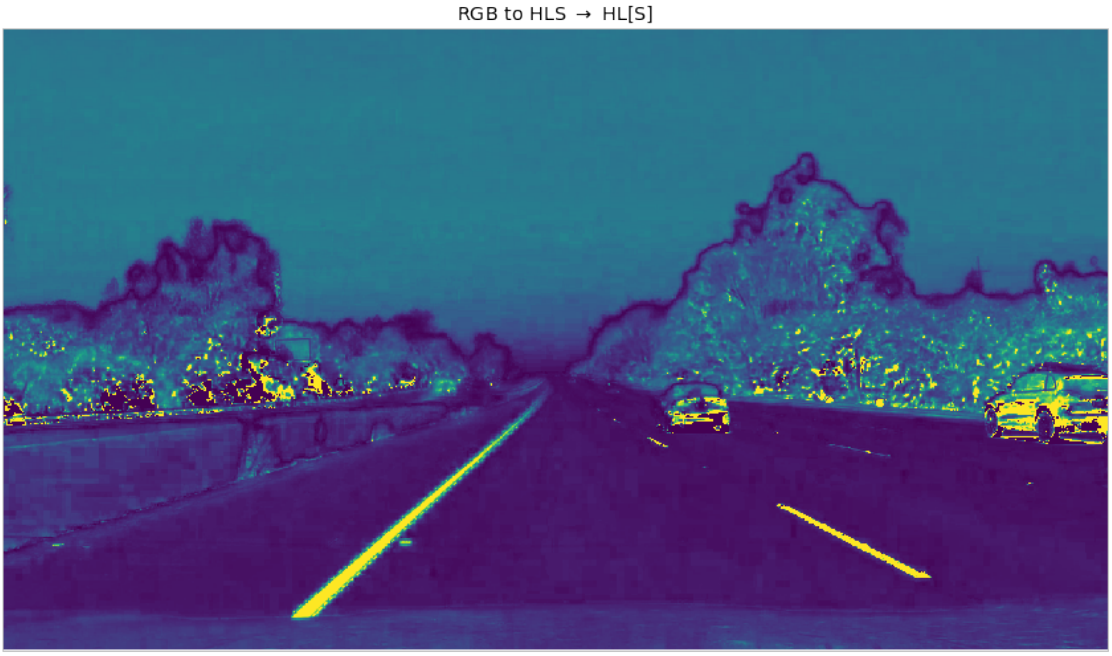
Thus, with one such node serving as the primary user-interactive root node for the digraph, setting its `.value` results in clearing the cached values for all descendants. I have not implemented this cache-clearing cascade behavior for any other types of root nodes (e.g., `Constant` nodes such as kernel matrices), since I figured I had to draw the line somewhere on this project. I also have not tested this approach with pipelines containing multiple `BaseImage` input nodes, but the behavior should work for those as well--only nodes which are direct descendants of the currently altered node will have their cache cleared, which is as it should be. Clearing the cache for a node twice is harmless.

```
In [4]: colorImage = cf.ColorImage()
        colorImage.value = frame
        colorImage.showValue();
```

→ ColorImage



```
In [5]: hls = cf.CvtColor(colorImage, cv2.COLOR_RGB2HLS)
        s = cf.ColorSplit(hls, 2)
        s.showValue();
```

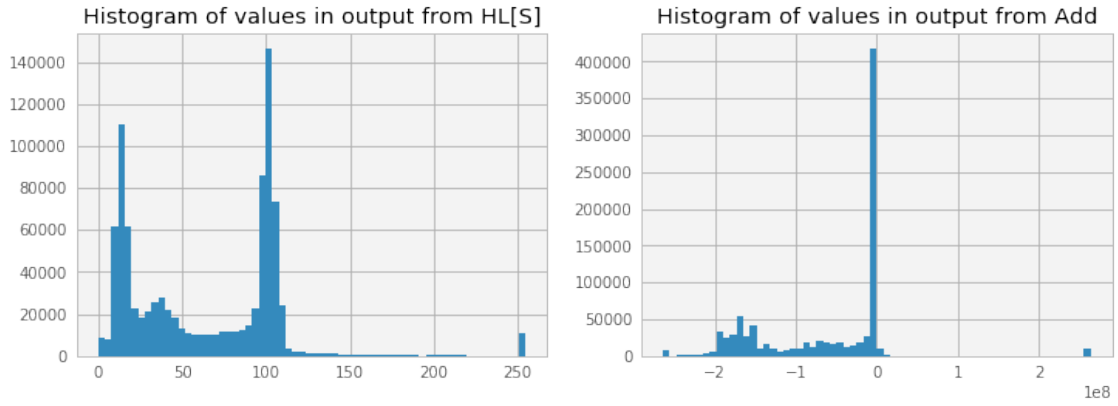


Some operator overloading is also implemented.

```
In [6]: # Re-implement the `cf.Expand` op,
        # for emphasizing outliers in a nonlinear way.
        x = cf.AsType(s, 'float64')
        centered = x - x.max() / 2
        neg = centered < 0
        pos = centered >= 0
        widenedFlippedSign = centered ** 4
        expanded = -abs(widenedFlippedSign & neg) + (widenedFlippedSign & pos)

        fig, axes = plt.subplots(ncols=2, figsize=(12, 4))
        for i, x in enumerate([s, expanded]):
            img = x.value

            ax = axes[i]
            ax.hist(img.ravel(), bins=64);
            ax.set_title('Histogram of values in output from %s' % x)
```



A second useful effect of this DAG structure is that requesting `object.value` on some leaf node propagates the execution of wrapped operations only up through direct ancestor nodes, without wastefully computing the results of side branches. This allows for the addition of speculative chains of operations that do not lead to the final result, for exploration purposes without slowing down the main execution, though there would be no reason to keep these in production. Analogously to the use of preprocessor directives in compiled languages, full-DAG operations take place only at instantiation time (and for some plotting and introspection purposes not necessary for executing the graph), not at execution.

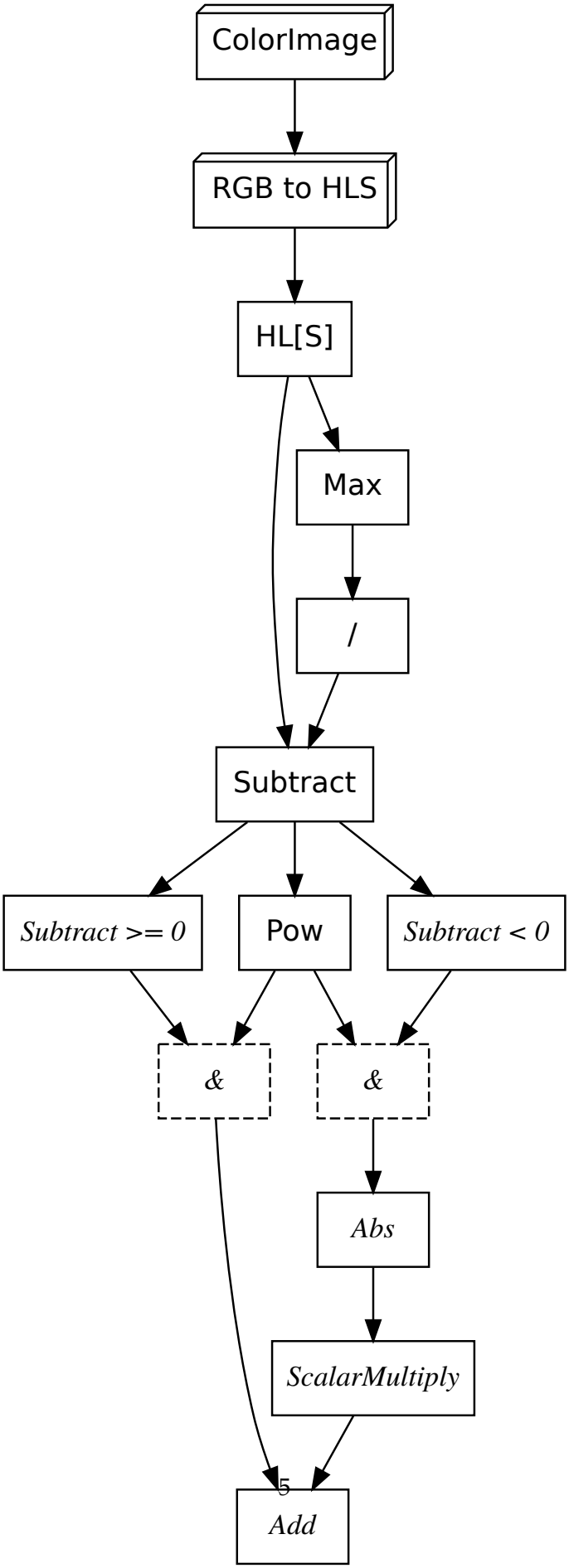
Subclassing `Op` is the `MultistepOp` class, which adds additional logic supporting nodes that wrap subgraphs of other nodes in convenient bundles. This became a primary motivation for the (perhaps inadvisable) creation of this library when I noticed that certain motifs of repeated OpenCV operations were useful in my pipeline.

Of course, such reused multi-step operations in OpenCV could be accomplished just fine through normal functions, but it would be difficult to introspect their structure and intermediate stages. This was the other major motivation for this library. Upon request, any node in the digraph can assemble an object representing the entire digraph in a convenient form (a combination of networkx and graphviz digraph objects). Courtesy of graphviz, this representation can be displayed in a Jupyter notebook, or written to various file formats.

We'll now use this to depict the DAG we've just constructed.

```
In [7]: expanded.draw(addKey=False)
```

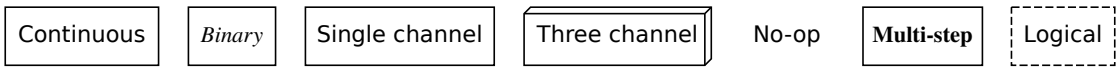
Out[7]:


ColorImage
RGB to HLS
HL[S]
Max
/
Subtract
Subtract >= 0
Pow
Subtract < 0
&
&
Abs
ScalarMultiply
Add

The key used for these diagrams is always as follows:

```
In [8]: # laneFindingPipeline.utils.bk()
        label, gv, nx = cf.misc.makeKeySubgraph()
        gv
```

Out[8]:

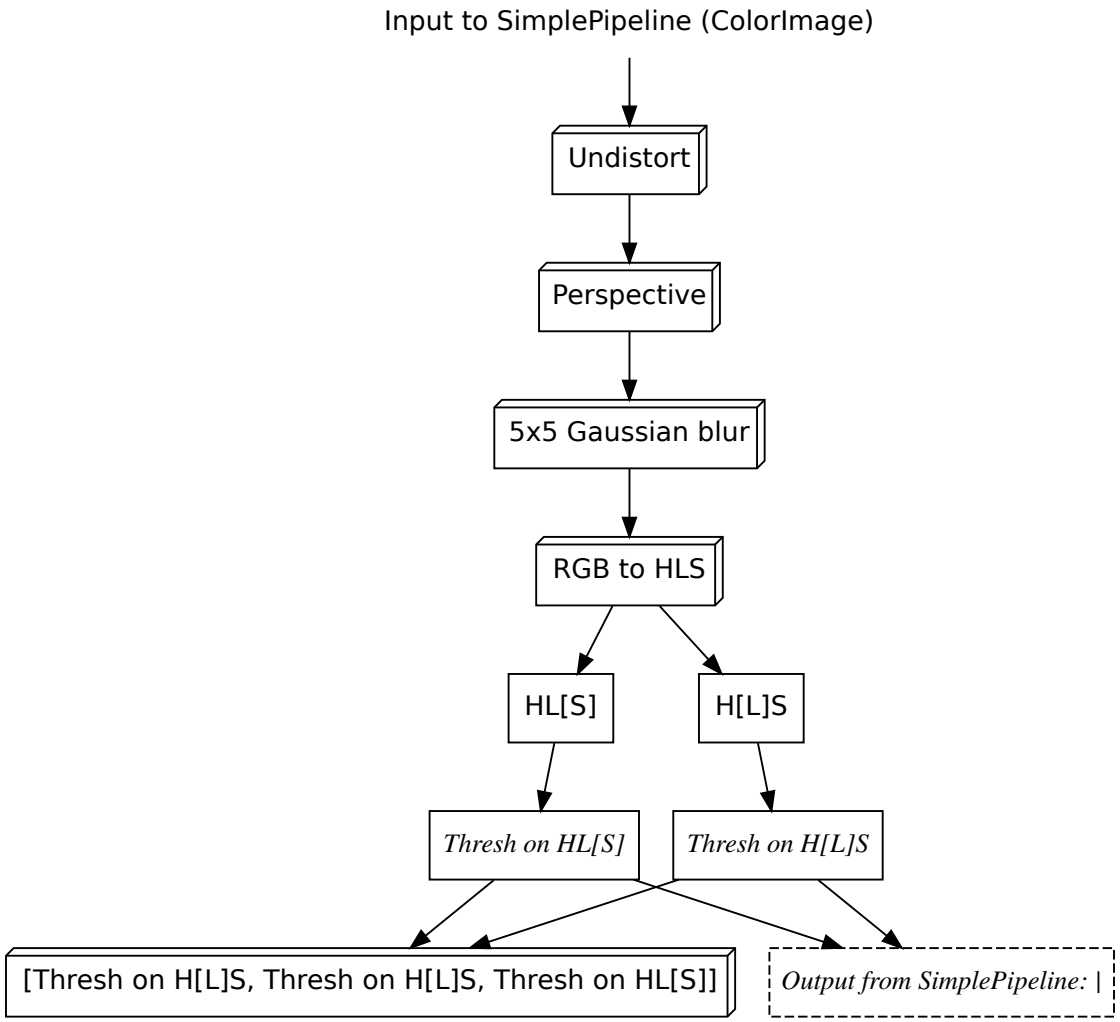


1.2 A simple pipeline

With significant effort invested to create this library, we can now use it to prototype some image pre-processing pipelines.

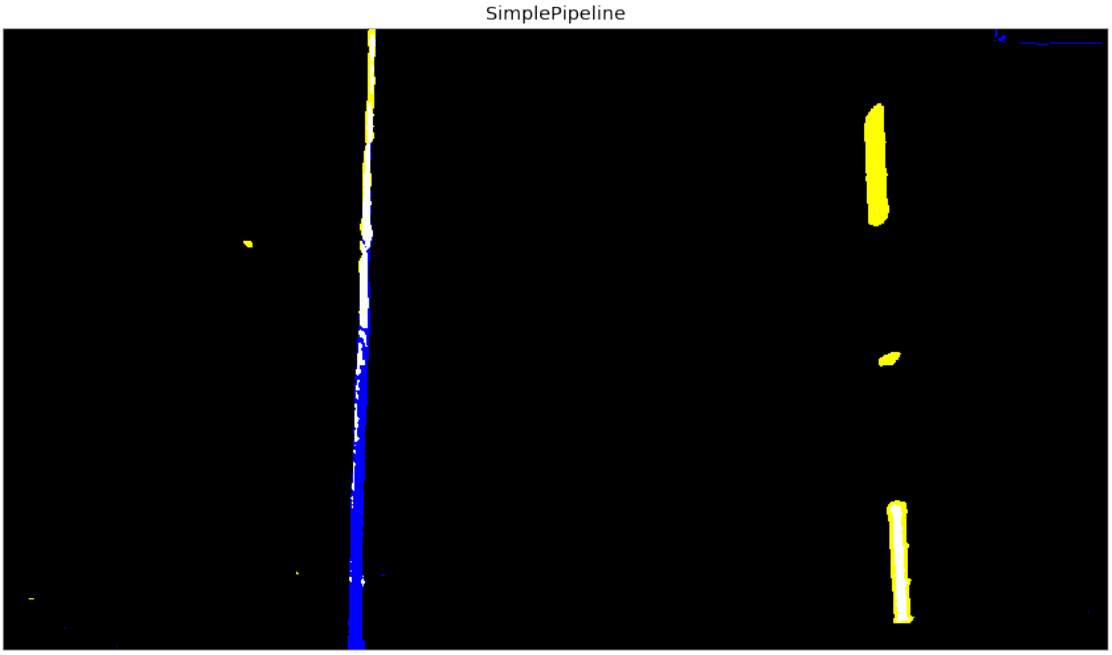
```
In [9]: pipeline = cf.SimplePipeline()
        pipeline.getSubgraph()
```

Out[9]:



This pipeline is fairly simple--we use the saturation channel to detect the yellow left line, and the lightness channel to detect the white right line. It works for most of the project video.

```
In [10]: linePixels = pipeline(frame)
         color = pipeline.colorOutput.value
         laneFindingPipeline.utils.show(color, title=pipeline);
```



the pipeline begins with correcting camera distortion and performing a perspective transform.

Brieftly, the undistortion attemps to uses `cv2.findChessboardCorners` to find a set of image points corresponding to known-planar object points for each of the supplied camera calibration images.

```
In [11]: src(cf.workers.UndistortTransformer.fitImg)
```

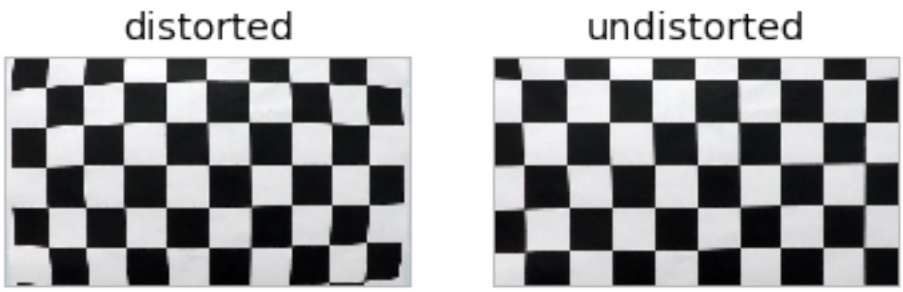
Out[11]:

Lines 19 throuh 26 of workers.py:

```
    def fitImg(self, img):
        if isinstance(img, str):
            img = cv2.imread(img)
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        self.imageShape = gray.shape[::-1]
        ret, corners = cv2.findChessboardCorners(gray, (self.nx, self.ny), None)
        if ret:
            self.imgp.append(corners)
```

```
In [12]: undistort = cf.workers.UndistortTransformer()
         distorted = cv2.imread('camera_cal/calibration1.jpg')
         undistort.fit()
         undistorted = undistort(distorted)

         fig, (ax1, ax2) = plt.subplots(ncols=2)
         laneFindingPipeline.utils.show(distorted, ax=ax1, title='distorted')
         laneFindingPipeline.utils.show(undistorted, ax=ax2, title='undistorted');
```



It then finds the camera parameters that best correct all of these distorted image points ot world points.

```
In [13]: src(cf.workers.UndistortTransformer.calcParams)
```

Out[13]:

Lines 48 throuh 52 of workers.py:

```
    def calcParams(self):
        objp = [self.singleObjP] * len(self.imgp)
        self.ret, self.mtx, self.dist, self.rvecs, self.tvecs = cv2.calibrateCamera(
            objp, self.imgp, self.imageShape, None, None
        )
```

This is implemented both in the worker class `UndistortTransformer` and in the node `Undistort`.

Likewise, a perspective transformation worker class uses `cv2.getPerspectiveTransform` during construction to find both forward and inverse transformation matrices, and `cv2.warpPerspective` to apply one or the other of these at call time.

```
In [14]: src(cf.workers.PerspectiveTransformer.__call__)
```

Out[14]:

Lines 132 throuh 140 of workers.py:

```
    def __call__(self, img, inv=False, img_size=None):
        if img_size is None:
            assert len(img.shape) < 4
            img_size = img.shape[:2][::-1]
        if inv:
            M = self.Minv
        else:
            M = self.M
        return cv2.warpPerspective(img, M, img_size, borderMode=cv2.BORDER_CONSTANT)
```

To find only relevant pixels in each channel, I use the complicated (but not `MultistepOp`) `CountSeekingThreshold` op.

```
In [15]: thresh = pipeline.getByKind(cf.CountSeekingThreshold)[0]
         propertySrc(thresh, 'value')
```

Out[15]:

Lines 175 throuh 216 of workers.py:

```
@cached()
def value(self):
    channel = self.parent().value
    goalCount = self.goalCount
    countTol = self.countTol

    def getCount(threshold):
        mask = channel > np.ceil(threshold)
        return mask, mask.sum()

    threshold = self.threshold

    under = 0
    over = 255
    getThreshold = lambda : (over - under) / 2 + under
    niter = 0
    while True:
        mask, count = getCount(threshold)
        if (
            abs(count - goalCount) < countTol
            or over - under <= 1
        ):
            break

        if count > goalCount:
            # Too many pixels got in; threshold needs to be higher.
            under = threshold
            threshold = getThreshold()
        else: # count < goalCout
            if threshold > 254 and getCount(254)[1] > goalCount:
                # In the special case that opening any at all is too bright, die early.
                threshold = 255
                mask = np.zeros_like(channel, 'bool')
                break
            over = threshold
            threshold = getThreshold()
        niter += 1

    out =  max(min(int(np.ceil(threshold)), 255), 0)
    self.threshold = out
    self.iterationCounts.append(niter)
    return mask
```

Basically, we set a `goalCount` of around 10000 pixels, then, rely on the fact that the number of thresholded pixels is monotonic in the current threshold to do a bifurcation search for the threshold that brings us within `countTol` of the goal.

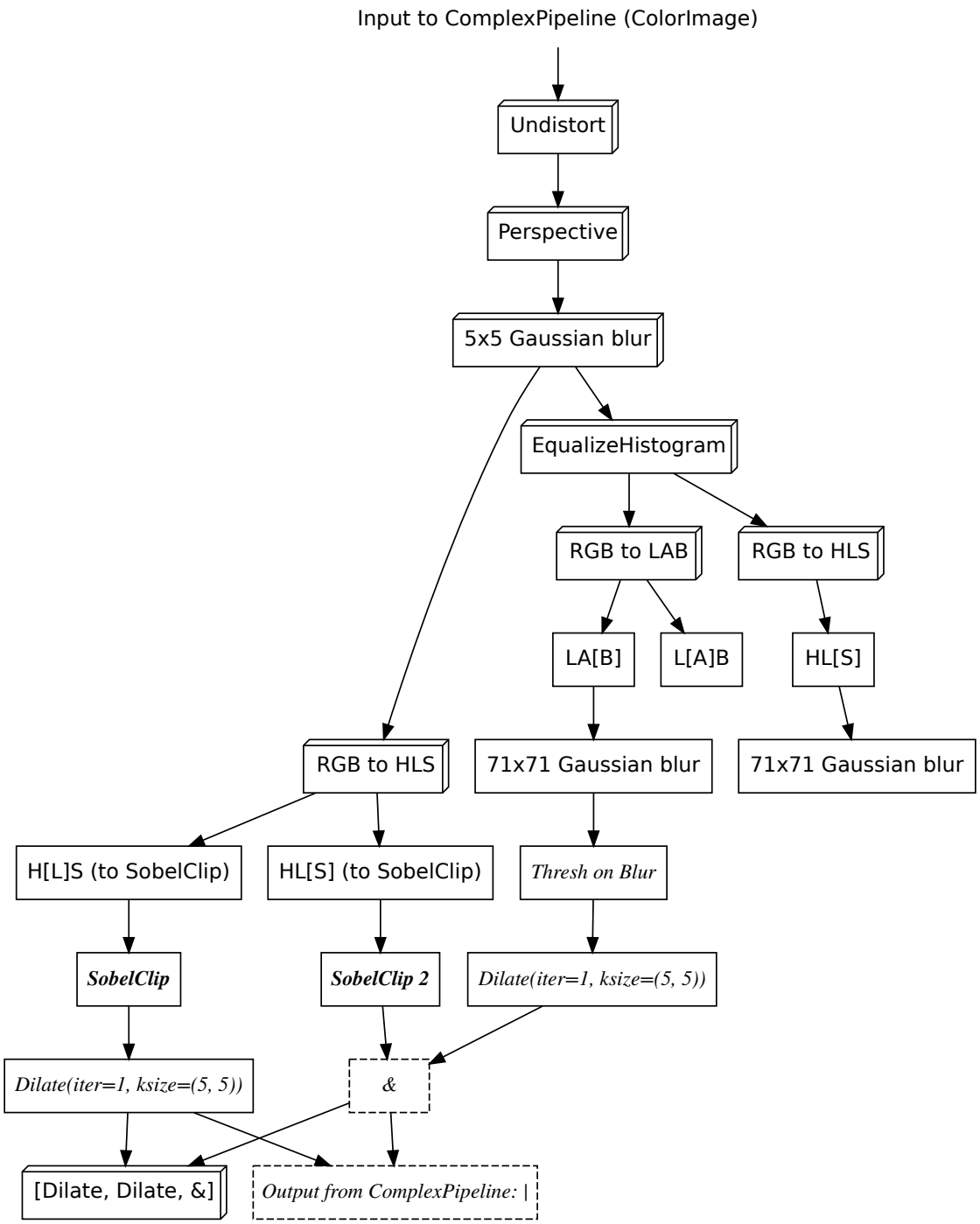
It occurs to me that such a dynamic thresholding might already be implemented in OpenCV, but, since other Sobel-only methods actually proved more useful than thresholding of any kind, I didn't pursue this further.

The problem with this or a fixed-threshold approach is that occasionally large splotches of brightness would come along in the monitored channel. For any reasonable `goalCount`, these splotches would quickly drive the threshold up to the maximum of 255, resulting in a mask empty of identified pixels.

1.3 A more complicated pipeline

```
In [16]: pipeline = cf.ComplexPipeline()
         # Initialize the input.
         pipeline(allFrames['harder_challenge'][-1]);
         pipeline.getSubgraph()
```

```
Out[16]:
```



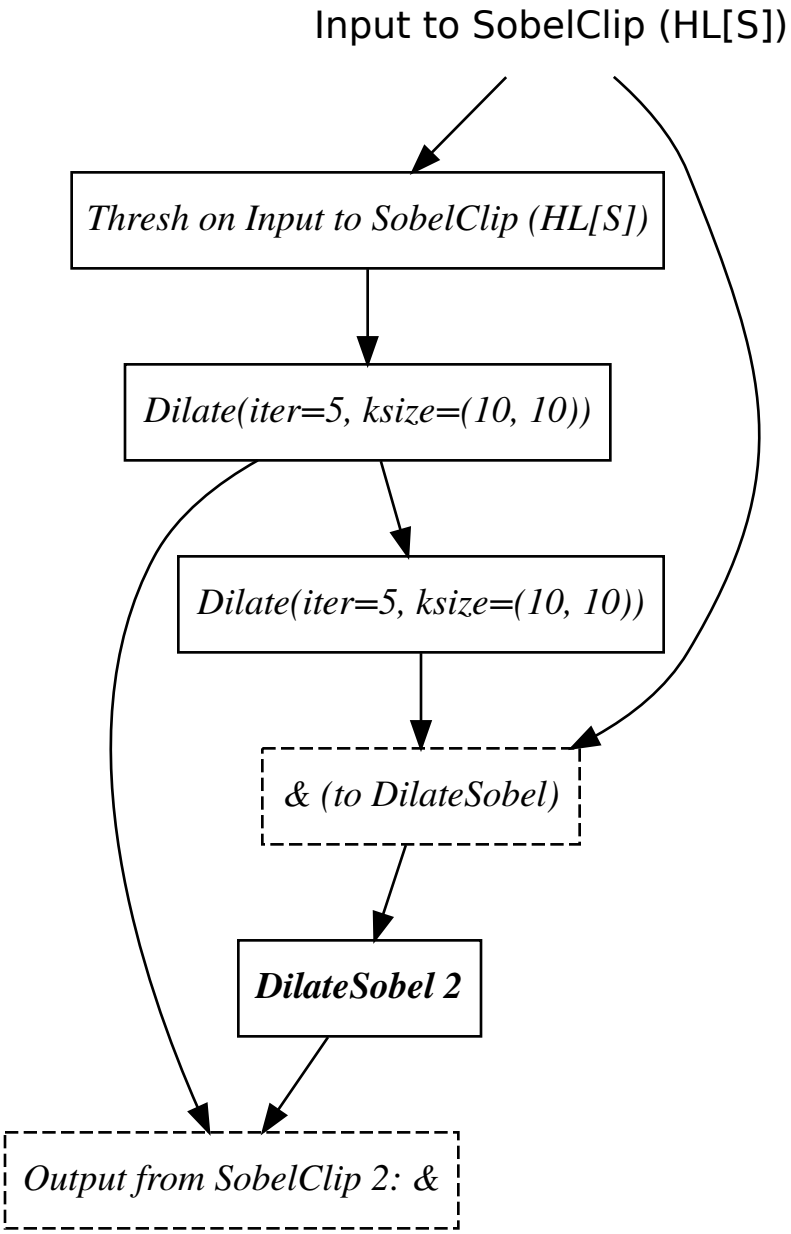
Though this pipeline was not used, it has a couple notable features. First, there are two leaf nodes--L[A]B and a blurred HL[S}--which do not contribute to the final output. These are included in some vlsualizations, but not computed for the purposes of finding actual lane markings.

1.3.1 The `SobelClip` `MultiOp`

Second, in addition to the `EqualizeHistogram` operation, which internally uses `cv2.equalizeHist`, there is a previously unseen operation called `SobelClip`. This is a subclass of `MultistepOp`.

```
In [17]: aSobelClip = pipeline.getByKind(cf.SobelClip)[0]
         aSobelClip.getSubgraph()
```

Out[17]:

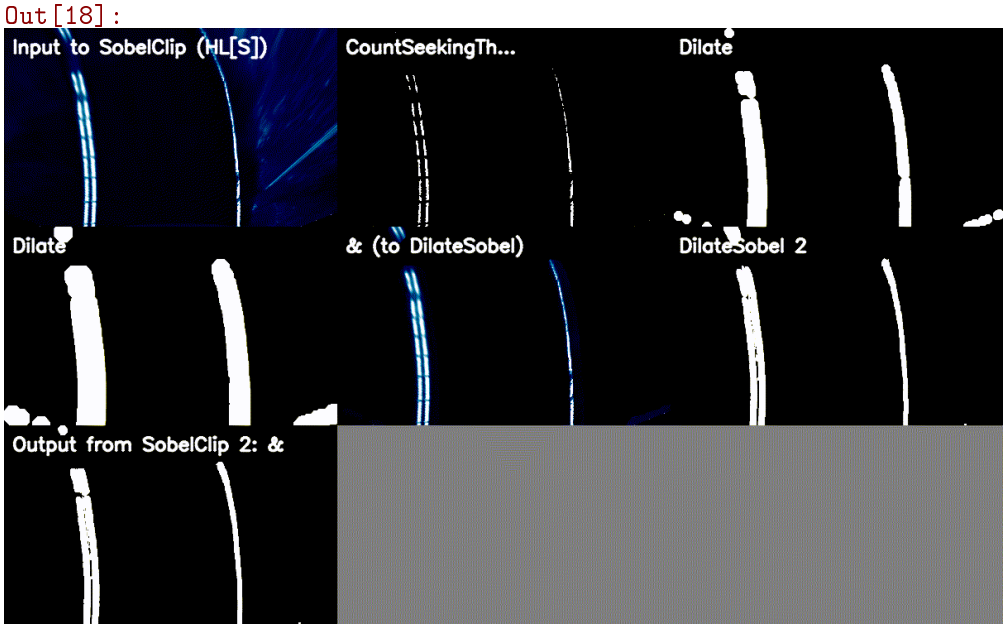


This operation works by first using our previously-discussed `CountSeekingThreshold` to detail an initial set of interesting pixels. This set is then slightly dilated to produce a restrictive mask for use in the final `&`, and dilated more to produce a permissive mask on the original data. this prevents the subsequent application of a `DilateSobel` operation (discussed below) from activating on irrelevant features (per one definition). Finally, the lesser dilation is used as a mask, alowing us to be relatively liberal with what we allow the wider mask to present to the `DilateSobel` for consideration.

(Note that outputs from `showOpClip` like the following, though static in PDF renderings of this notebook, are animated in the notebook visible on Github.)

```
In [18]: showOpClip(aSobelClip)
```

Out[18]:



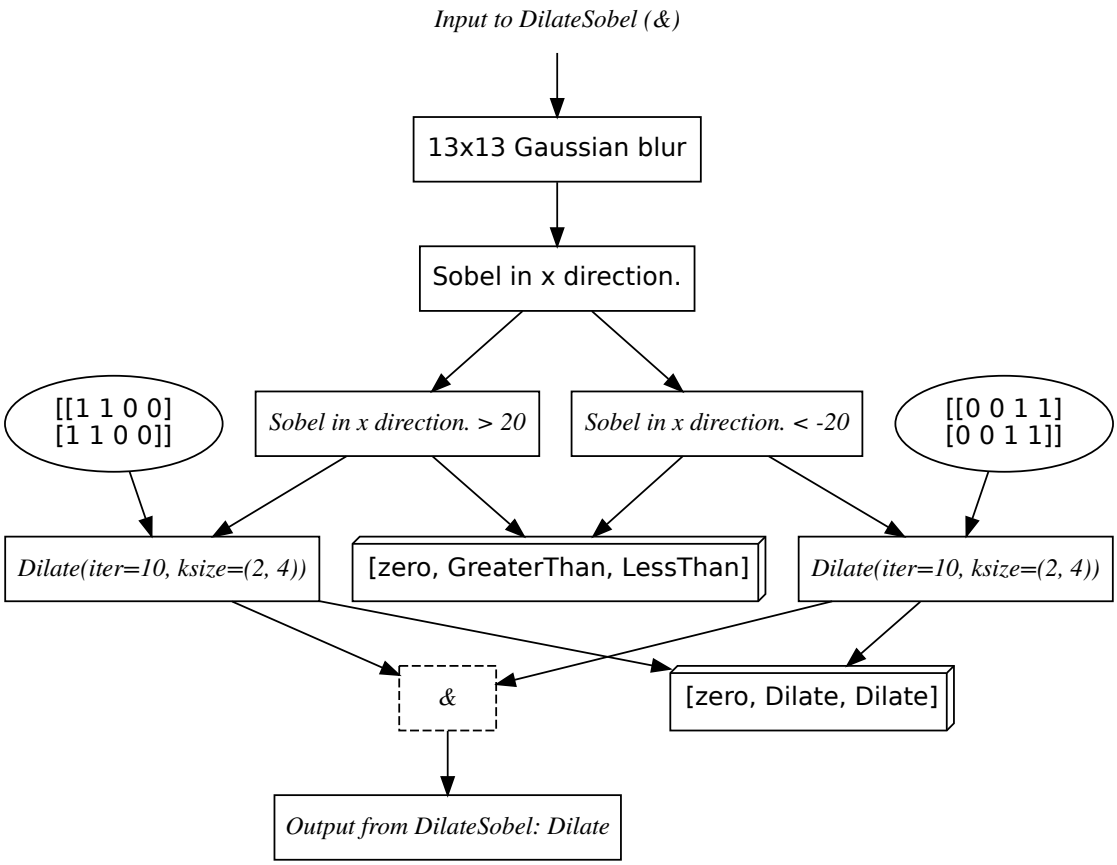
While this might sometimes produce some useful results, as in the example above, its reliance on on a threshold makes it vulnerable to large splotches of brightness or darkness in the observed channel. My later work made me question what value these dilated-threshold masks were adding to the underlying Sobel-based method.

1.3.2 The Dilate Sobel `MultistepOp`

On the other hand, one composite operation that I did find repeatedly useful was a combination of two asymmetric Sobel operations.

```
In [19]: aDilateSobel = pipeline.getByKind(cf.DilateSobel)[0]
         # show ndarray kernels explicitly
         for dilate in aDilateSobel.getMembersByType(cf.Dilate):
             for parent in dilate.parents:
                 if isinstance(parent, cf.Constant):
                     # Show the kernels in the diagram, but not in the preview clip.
                     parent.hidden = parent.isVisualized = False
                     if isinstance(parent.value, np.ndarray):
                         parent.nodeName = '%s' % parent.value
         aDilateSobel.getSubgraph()
```

Out[19]:



Note that, here, I've unhidden two constant dilation kernels.

Sobel in the x direction returns a continuous-valued array of positive and negative values, indicating locations where the x-derivative of the image is positive or negative, respectively. By using two thresholds, we separately extract these two rising- and falling-edge signals. Since lane lines are generally always a fixed number of pixels wide in our images, **we generally want only vertical rising edges that are quickly followed by falling edges** (for "ridges"; this is reversed for "troughs", which is how lane lines might manifest in some channels).

We can extract only such edge combinations by differentially dilating the our two thresholded sobel signals. Since dilation can be thought of as pulling a pixel value from neighbors indicated by nonzero locations in the centered kernel, the two constant kernels shown in the diagram above have the effect of dilating these found pixels exclusively to one side or the other. When rising edges are dilated to the right, and falling to the left, This filter discovers vertical ridges of a particular width. This is visible in the color composite images below, which show the two masks before and after this dilation, as well as the intersection that appears.

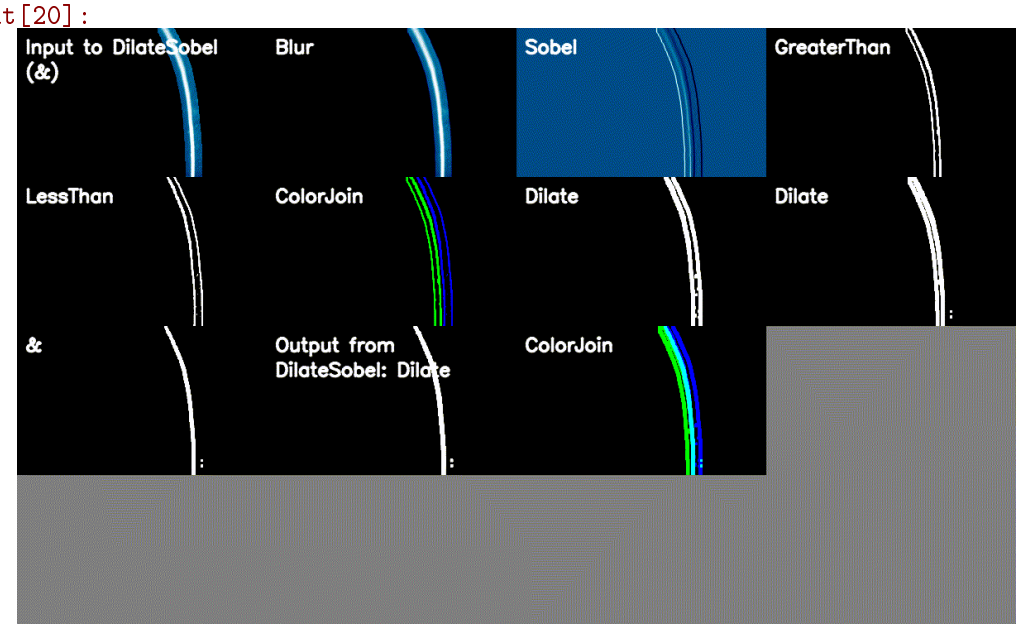
In this particular example, the input to the `DilateSobel` comes from an enclosing `SobelClip` operation, so the true edge is enclosed in a wide-masked bounding tube. While the initial Sobel convolution of the `DilateSobel` picks up these outer edges as well, the intersection of the differentially dilated positive and negative thresholds correctly excludes these outer edges as being part of a structure of the incorrect width.

In the next pipeline, I do use `DilateSobel` directly without `SobelClip`, and so this outer bounding tube does not take effect.

```
In [20]: showOpClip(aDilateSobel)
```

```
/mnt/wdblue/Dropbox/Projects/Lane Lines/cvflow/misc.py:420: UserWarning: Using a non-unity scali
  warn('Using a non-unity scaling for float image data. This might cause flashing in output.')
```

Out[20]:



1.4 Full pipeline

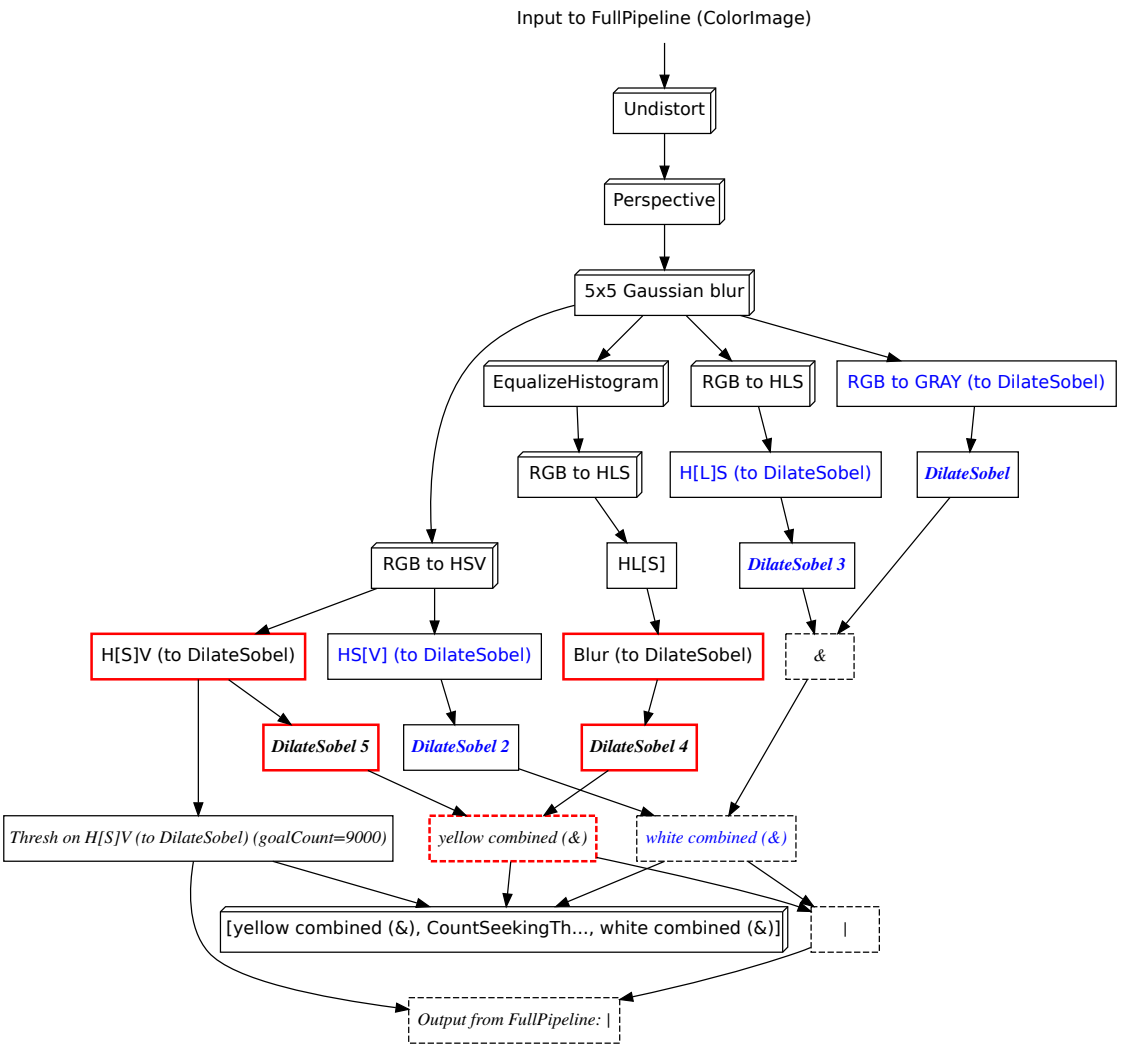
Using methods from these last two pipelines, I set out to design a preprocessing pipeline in a more principled way. Namely, My strategy was to produce several pixel-finders focused on identifying a particular line in a "permissive" way (that is, they err on the side of finding too many pixels) and then take the intersection (`&`) of these masks to produce a "restrictive" mask focused on that feature. I did this separately for the left (yellow solid) and right (white dashed) lane markings.

Finally, I took the union (`|`) of these restrictive masks to produce the final output of the preprocessing pipeline.

In the diagram below, I style some nodes focused on discovering these left and right markings with red and blue annotations, respectively (it is possible for some nodes to be used for both). Broadly speaking, I use color features for finding the yellow marking, and brightness features to find the white marking.

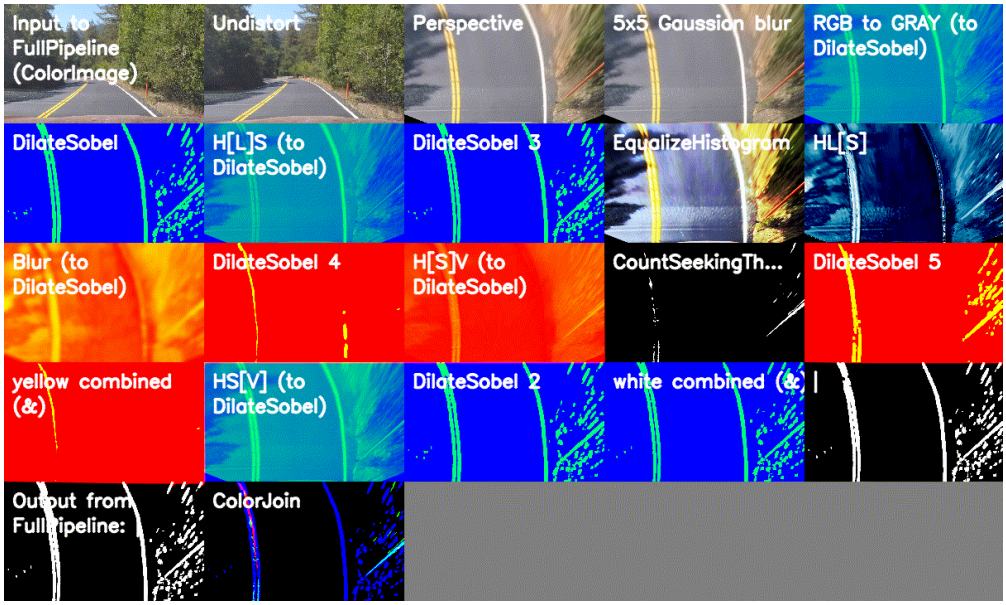
```
In [21]: pipeline = cf.FullPipeline()
         pipeline.getSubgraph()
```

Out[21]:



In [22]: showOpClip(pipeline)

Out[22]:



2 Curve fitting

In the end, our preprocessing pipeline (usually) transforms raw camera frames to masks on relevant lane-marking pixels.

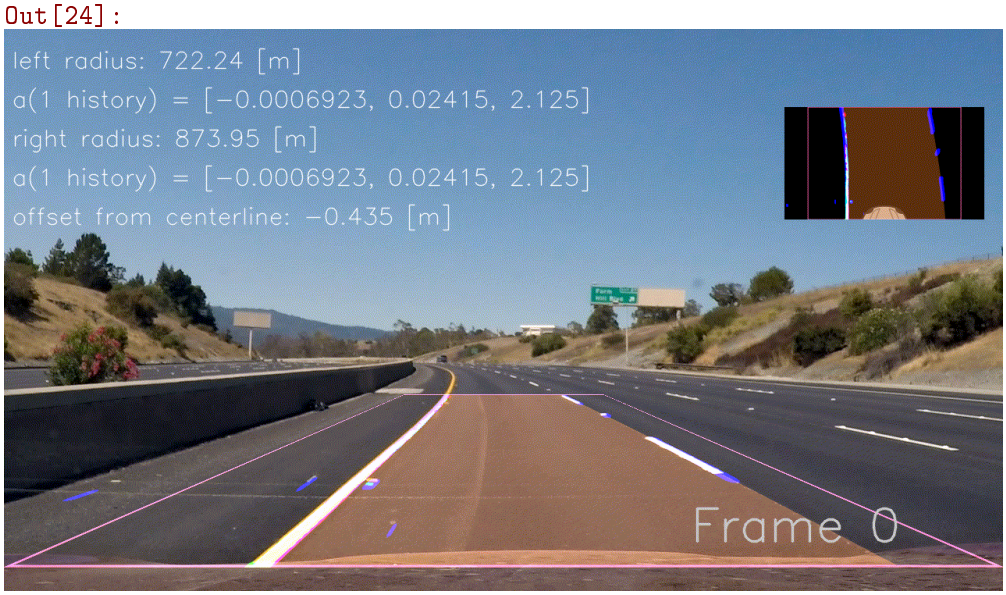
```
In [23]: show = laneFindingPipeline.utils.show
         fig, (left, right) = plt.subplots(ncols=2, figsize=(16, 6));
         show(frame, ax=left, title='camera input');
         show(pipeline(frame), ax=right, title='preprocessed');
```

The goal now is to fit two polynomials to this data, and extract from these curves some geometric information which would presumably be useful for control tasks such as lane-keeping.

```
In [24]: laneFinder = laneFindingPipeline.LaneFinder(colorFilter=pipeline)

         laneFinder.process(
             allFrames['project'][:32], 'doc/images/project_results.gif', tqdmKw=dict(pbar=False
             showCentroids=False,
         )
```

Out[24]:



2.1 Assigning pixels to the right and left markings

It was suggested to use some combination of three possible methods to find a subset of pixels on which to regress our lane-defining polynomials. We choose from: 1. For each horizontal slice of the image, sum over rows, then look for peaks in the resulting signal. Stitch these peaks together across slices to get a set of horizontal box centers. For boxes of a given width, take all the pixels within identified by the preprocessing filter. 2. Similar, but with convolutions. I chose to use a gaussian kernel. For a given horizontal slice of the preprocessed image, we sum over rows, and then, instead of explicitly searching for peaks, we search for the peak in the convolution of this signal with a chosen filter. This method is slightly more robust to noise. 3. Given a previously fitted polynomial, search in a fixed margin to the left and right for pixels identified by the preprocessor. Of the three, this method most imposes the strongest assumption that the shape of the lane lines will not change drastically from frame to frame.

The convolutional pixel finder begins by establishing a kernel (`window`) to be swept over the image, and determining the locations of nonzero pixels in the input. We use `initialGuessMarking` to generate naive starting guesses in for x-locations of the curves, in the centers of the left and right sides of the image.

```
In [25]: convolutionalMarkingFinder = laneFindingPipeline.ConvolutionalMarkingFinder()
         src(convolutionalMarkingFinder.update, end=42)
```

Out[25]:

Lines 175 throuh 342 of laneFindingPipeline.py:

```
    def update(self, image, _recursionDepth=0):
        """Find lane markings in preprocessed image.

        Parameters
        ----------
        image : ndarray
```

```
        (Ignore _recursionDepth)

        Sets attributes
        ---------------
        self.markings : list
            The found LaneMarking objects
        self.windowCentroids : list of lists
            Column (x) locations of the found window centroids
        """
        windowWidth = self.windowWidth
        windowHeight = self.windowHeight
        gaussianRadius = self.gaussianRadius
        nlevels = int(image.shape[0] / windowHeight)

        # Refer to a common set of nonzero pixels.
        nonzero = image.nonzero()
        nonzeroy = np.array(nonzero[0])
        nonzerox = np.array(nonzero[1])

        ## Store some things per-marking, per-level.
        # Store the window centroid positions per level.
        windowCentroids = [[None] * nlevels for m in self]
        # Store the included pixel indices.
        lane_inds = [[None] * nlevels for m in self]

        # Create our window template that we will use for convolutions.
        if self.windowType == 'gaussian':
            # Gaussian window
            window = np.exp(-np.linspace(-gaussianRadius, gaussianRadius, windowWidth)**2)
        else:
            # Box window.
            window = np.ones(windowWidth)

        # Get an initial guess for the centers.
        centers = [initialGuessMarking(image.shape[0]) for initialGuessMarking in self]
```

We then iterate over a small number of horizontal slices of the images (`levels`). In each, we 1. Sum over rows in the slice to create a 1D row-averaged signal. 2. If there are any nonzero convolution values in a restricted search window around the previous center (or initial guess), choose the new center to be at the maximum (else, leave the center unchanged). 3. Accumulate from each level the pixels inside the optimal convolution window.

In step 2, if no convolution pixels are nonzero, I previously tried using a linear extrapolation of the centers of the last two levels (if available). This is helpful, for instance, when there are large gaps in the broken white line. However, this sometimes backfires when a blob of pixels beside the marking pushes the window to the left and is followed by such a gap, which then produces a strongly wrong linear-projection guess for the next-level center.

```
In [26]: src(convolutionalMarkingFinder.update, start=42, end=95)
```

Out[26]:
Lines 217 throuh 437 of laneFindingPipeline.py:

```
        # Iterate over horizontal slices of the image.
        for level in range(nlevels):

            # Sum over rows in this horizontal slice
            # to get the scaled mean row.
            image_layer = np.sum(image[
                int(image.shape[0]-(level+1)*windowHeight)
                :
                int(image.shape[0]-level*windowHeight)
                ,:
            ], axis=0)

            # Convolve the window into the signal.
            conv_signal = np.convolve(window, image_layer, mode='same')

            # Find the best left centroid by using past center as a reference.
            searchBoxes = self._getSearchBoxes(centers, image, level, nlevels)

            # Find the best centroids by using past centers as references.
            # TODO: If the bracketed portion is all-zero, consider that maybe this method has fa
            for i in range(len(self)):
                lo, hi = searchBoxes[i]
                bracketed = conv_signal[lo:hi]
                if bracketed.any():
                    # If the convolution picked up something in this bracket,
                    # make that the new center.
                    center = np.argmax(bracketed) + lo
                else:
                    # Do nothing.
                    center = centers[i]

                # Save centers for the next level.
                centers[i] = center

                ## Save our results for this level.
                windowCentroids[i][level] = center

                # Identify window boundaries in x and y (and right and left).
                win_y_low = image.shape[0] - (level+1)*windowHeight
                win_y_high = image.shape[0] - level*windowHeight
                win_x_low = center - windowWidth / 2
                win_x_high = center + windowWidth / 2

                # Identify the nonzero pixels in x and y within the window.
```

```
            good_inds = (
                (nonzeroy >= win_y_low) & (nonzeroy < win_y_high)
                &
                (nonzerox >= win_x_low) & (nonzerox < win_x_high)
            ).nonzero()[0]

            # Insert these indices in the lists.
            lane_inds[i][level] = good_inds
```

Finally, we use the found indices to extract the actually pixel locations, and use these to regress two `LaneMarking` objects, and save the found centers for plotting purposes.

```
In [27]: src(convolutionalMarkingFinder.update, start=95)
```

Out[27]:
Lines 270 throuh 394 of laneFindingPipeline.py:

```
        # Concatenate the arrays of indices.
        all_lane_inds = [np.concatenate(ii) for ii in lane_inds]

        # Extract line pixel positions.
        X = [nonzerox[ii] for ii in all_lane_inds]
        Y = [nonzeroy[ii] for ii in all_lane_inds]

        # If the location arrays are somehow totally empty, revert to default vertical lines.
        for i in range(len(self)):
            if len(X[i]) == 0:
                assert len(Y[i]) == 0
                X[i] = np.array((self.hintsx[i],), X[i].dtype)
                Y[i] = np.zeros((1,), Y[i].dtype)

        # Assemble 2-row point arrays.
        markingsPoints = [
            np.stack([x, y])
            for (x, y) in zip(X, Y)
        ]

        # Generate the new lane markings.
        self.markings = [
            LaneMarking(points)
            for points in markingsPoints
        ]

        self.windowCentroids = windowCentroids

        self.postUpdateQualityCheck(image, _recursionDepth)
```

At the moment, this is followed up by a set of quality checks used to decide whether these markings should be kept, or replaced with default guesses. If the checks fail (return `False`), they

reset the last-guess storge (`self.__getitem__`). We then repeat the update with the new new (default) guesses. A better use for this check is dicussed in the conclusion section below.

```
In [28]: src(laneFindingPipeline.MarkingFinder.postUpdateQualityCheck)
```

Out[28]:
Lines 117 throuh 124 of laneFindingPipeline.py:

```
 def postUpdateQualityCheck(self, image, _recursionDepth, maxRecursion=1):
     # Check some quality metrics on the found lines.
     # If they're found lacking, the checker will replace them
     # with the default guesses. We can then try the update again.
     if not self.evaluateFitQuality():

         if _recursionDepth < maxRecursion:
             self.update(image, _recursionDepth=_recursionDepth+1)
```

The checks themselves consist of 1. ensuring that sufficiently many nonzero pixels were actually found, 2. ensuring that the two radii of curvature aren't drastically different, and 3. ensuring that the two fits haven't collappsed into the same curve.

Additionally, I generate an overall quality score, but this is poorly tuned of dubious use.

```
In [29]: src(laneFindingPipeline.MarkingFinder.evaluateFitQuality)
```

Out[29]:
Lines 65 throuh 115 of laneFindingPipeline.py:

```
 def evaluateFitQuality(self, lowPointsThresh=1, radiusRatioThresh=500):
     self.failedLaneMarkings = []

     ## Check single-marking quality indicators.
     # Some points were actually found?
     acceptables = [True] * len(self)
     for i, laneMarking in enumerate(self):

         if len(laneMarking.x) <= lowPointsThresh:
             acceptables[i] = False

     ## Check paired quality indicators:
     # Radii aren't drastically different?
     assert len(self) == 2
     left, right = self
     rads = [left.radius, right.radius]
     rads.sort()
     radRat = rads[1] / rads[0]
     if radRat > radiusRatioThresh:
         acceptables = [False] * 2

     # The two fits aren't actually the same?
     a = left.worldFit
```

```
        b = right.worldFit
        assert len(a) == len(b)
        # Different thresholds are appropriate for different coeffients.
        thresholds = [10**-(k+1) for k in range(len(a))][::-1]
        if not (np.abs(a - b) > thresholds).any():
            acceptables = [False] * 2

        # Assign quality scores.
        qualityFactors = 10, 1, 1; norm = sum(qualityFactors)
        for i, laneMarking in enumerate(self):
            mse = laneMarking.mse
            n = len(laneMarking.x)
            qualityVec = (
                n / 9001.,
                100. / (mse if (mse != 0 and n > 2)  else np.inf),
                3.0 / radRat,
            )

            laneMarking.quality = sum([q*fac/norm for (q, fac) in zip(qualityVec, qualityFactors
            laneMarking.qualityVec = qualityVec

        for i, acceptable in enumerate(acceptables):
            if not acceptable:
                self.failedLaneMarkings.append(self.markings[i])
                self.markings[i] = self.getHintedLaneMarking(i)

        # Report to the caller whether all laneMarkings passed inspection.
        return np.all(acceptables)
```

2.2 Fitting `LaneMarking` objects

The `LaneMarking` object holds polynomial fits to provided (y,x) data, and calculates radius of curvature on demand. At the moment, I simply use the same regression method on data with transformed units to produce `worldFit` in addition to `fit`. Really, this should be analytically computed.

```
In [30]: src(laneFindingPipeline.LaneMarking.__init__)
```

Out[30]:
Lines 430 throuh 483 of laneFindingPipeline.py:

```
 def __init__(self, points=None,
        order=2,
        ransac=False, lamb=.5,
        smoothers=(lambda x: x, lambda x: x)
        ):
        """
        Parameters
        ----------
        points : ndarray, shape (2, n)
```

```
        Associated pixel locations
    order : int, optional
        Order for the fitting polynomial. Probably don't want to mess with this.
    ransac : bool, optional
        Whether to use RANSAC or normal polynomial regression.
    lamb : float, optional
        L2 regularization coefficient.
    smoothers : tuple of two callables
        Functions that take in a coefficient vector and return a vector
        of the same size, perhaps with some stateful smoothing applied.

    """

    # Default case for no-argument construction
    self.imageSize = (720, 1280)
    if points is None:
        self.initialized = False
        points = [[self.imageSize[1]/2]*10, np.linspace(0, self.imageSize[0], 10)]
    else:
        self.initialized = True

    points = np.asarray(points)
    assert points.shape[0] == 2
    self.x, self.y = x, y = points
    self.order = order
    self.xm_per_pix = 3.6576/628.33
    self.ym_per_pix = 18.288/602
    self.radiusYeval = 720
    self.smoothers = smoothers
    self.quality = float(len(x))
    self.qualityVec = [None]*3

    # TODO: Use RANSAC to do the fitting.
    # Construct the fits by hand if a vertical line is given,
    # to avoid RankWarning from numpy.
    if len(set(x)) == 1:
        self.fit = np.zeros((order+1,))
        self.fit[-1] = x[0]
        self.worldFit = np.zeros((order+1,))
        self.worldFit[-1] = x[0] * self.xm_per_pix
        self.mse = 0
    else:
        self.fit = self.regressPoly(y, x, order, ransac=ransac, lamb=lamb)
        self.worldFit = self.regressPoly(y * self.ym_per_pix, x * self.xm_per_pix, order, ra
        self.mse = np.mean((self() - x)**2)
```

I also allow for calculating the radius of curvature of these lane markings by leveraging the `np.polyder` method, evaluating at the point nearest the camera. So, technically, we're prepre-

pared for any order of fitted polynomial, though I suspect that anything over cubic wouldn't help anything.

```
In [31]: propertySrc(laneFindingPipeline.LaneMarking(), 'radius')
```

Out[31]:
Lines 564 throuh 575 of laneFindingPipeline.py:

```
    @property
    def radius(self):
        """Radius of curvature near the camera, in meters."""
        yeval = self.radiusYeval * self.ym_per_pix
        fit = np.poly1d(self.worldFit)
        xp = np.polyder(fit,  m=1)
        xpp = np.polyder(fit, m=2)
        num = (1 + xp ** 2)(yeval)
        den = abs(xpp(yeval))
        if den == 0:
            return np.inf
        return num ** (3./2) / den
```

I do include a Scikit-Learn RANSAC regressor as an option, but do not use it since it requires some tuning which I don't have time to do now (but see the conclusion section below).

```
In [32]: src(laneFindingPipeline.LaneMarking.regressPoly)
```

Out[32]:
Lines 485 throuh 525 of laneFindingPipeline.py:

```
    @staticmethod
    def regressPoly(x, y, order, ransac=False, lamb=0.5):
        """Regress a polynomial for y = f(x).

        Parameters
        ----------
        x : iterable, length n
            Independent/predictor variable
        y : iterable, length n
            Dependent/response variable
        order : int
            Polynomial order
        ransac : bool, optional
            Whether to use RANSAC robust regression
        lamb : float, optional
            L2 regularization coefficient

        Returns
        -------
        fit : ndarray, length order+1
            The monomial coefficients in decreasing-power order.
```

```
    """
    # scikit-learn.org/stable/auto_examples/linear_model/plot_robust_fit.html
    # scikit-learn.org/stable/modules/generated/sklearn.linear_model.RANSACRegressor.html
    if lamb == 0:
        if not ransac:
            return np.polyfit(x, y , order)
        else:
            estimator = RANSACRegressor(random_state=42, min_samples=.8, max_trials=300)
            model = make_pipeline(PolynomialFeatures(order), estimator)
            model.fit(x.reshape(x.size, 1), y)
            return model._final_estimator.estimator_.coef_[::-1]
    else:
        A = np.stack([np.asarray(x).ravel()**k for k in range(order+1)[::-1]]).T
        n_col = A.shape[1]
        fit, residuals, rank, s = np.linalg.lstsq(
            A.T.dot(A) + lamb * np.identity(n_col),
            A.T.dot(y)
        )
        return fit
```

2.3 Meters per pixel

To determine the correct right y and x meters/pixel conversions, I took a well-behaved frame from the main project video, measured the relevant distances, and compared these to the California lane marking spacing standards. These say that these white broken highway markings should be 12 feet long (with a small dot centered in the 36-foot gap between markings), and that lanes are generally 12 feet wide.

For good measure, I also determined the width in pixels of a typical 80-inch car, and displayed a little car decal on the inset image in the main output video.

3 Smoothing fits over time

To hedge against sharp changes in the fit which aren't rejected by the quality checks above, we use a simple box filter to average fits over time for use in the actual plotting. This is implemented in terms of a general smoothing window, to allow for future use with the `WeightedSmoother` class, if I can devise a reasonable way to give fits continuously-valued quality scores.

```
In [33]: import smoothing
```

```
In [34]: src(smoothing.Smoother, end=-4)
```

Out[34]:

Lines 4 throuh 8 of smoothing.py:

```
class Smoother:
    """Running average"""
```

```
    def __init__(self, historySize=10):
        self.history = deque(maxlen=historySize)
```

```
In [35]: src(smoothing.WindowSmoother)
```

Out[35]:

Lines 14 throuh 24 of smoothing.py:

```
class WindowSmoother(Smoother):
    """Running average weighted by some window function"""

    def __call__(self, x):
        self.history.append(x)
        l = len(self.history)
        indices = range(0, -l, -1)
        normalizer = sum([self.window[i] for i in indices])
        return sum([
            self.window[i] * self.history[i] for i in indices
        ]) / normalizer
```

```
In [36]: src(smoothing.BoxSmoother)
```

Out[36]:

Lines 26 throuh 31 of smoothing.py:

```
class BoxSmoother(WindowSmoother):
    """Running average weighted by a uniform window function"""

    def __init__(self, historySize=10):
        WindowSmoother.__init__(self, historySize=historySize)
        self.window = np.ones((historySize,))
```

4 Final result

For each frame, we use the preprocessing pipeline created with `cvflow` to undistort, perspective-transform, and threshold images. We then use code in `LaneFinder` to create some `LaneMarking` objects.

Finally, we use `LaneFinder.draw` to found lane fill, convolutional search windows, etc. first onto a small inset figure, then (afer calling `perspectiveTransformer(inset, inv=True)` to perform the inverse transformation for painting over the main video with `cv2.addWeighted`.

Below, we'll run this complete pipeline on each of the provided test videos.

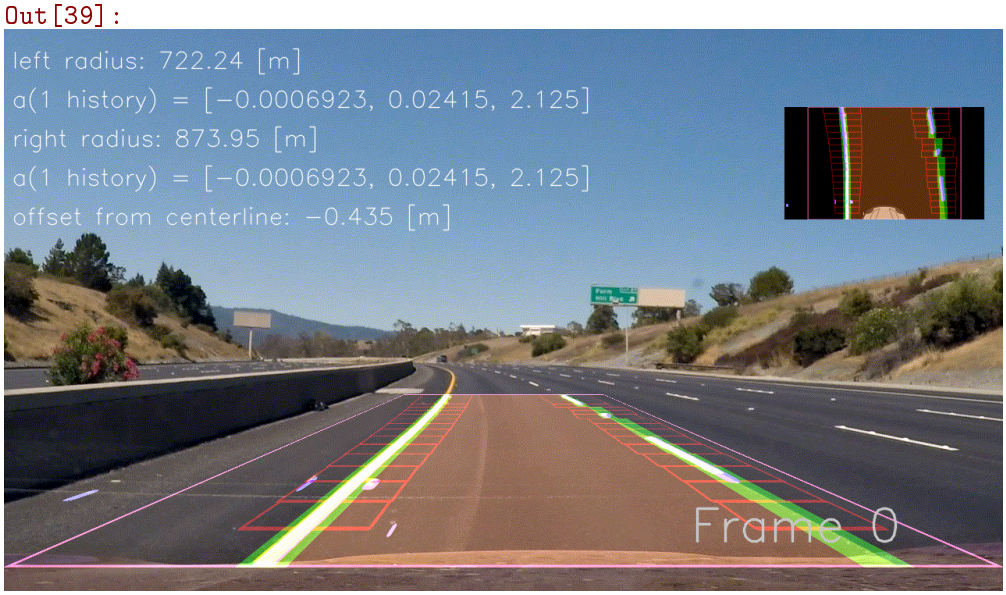
```
In [44]: # vids = {}
         # for key in 'project', 'challenge', 'harder_challenge':
         #     laneFinder = laneFindingPipeline.LaneFinder(colorFilter=pipeline)
         #     vid = laneFinder.process(
         #         allFrames[key], 'doc/images/%s_full_results.mp4' % key,
         #     );
         #     vids[key] = vid
```

I've uploaded these results to youtube for the main project video, challenge, and the harder challenge.

In [45]:
```
gifs = {}
for key in 'project', 'challenge', 'harder_challenge':
    laneFinder = laneFindingPipeline.LaneFinder(colorFilter=pipeline)
    gif = laneFinder.process(
        allFrames[key], 'doc/images/%s_full_results.gif' % key,
        maxFrames=32, tqdmKw=dict(pbar=False)
    );
    gifs[key] = gif
```

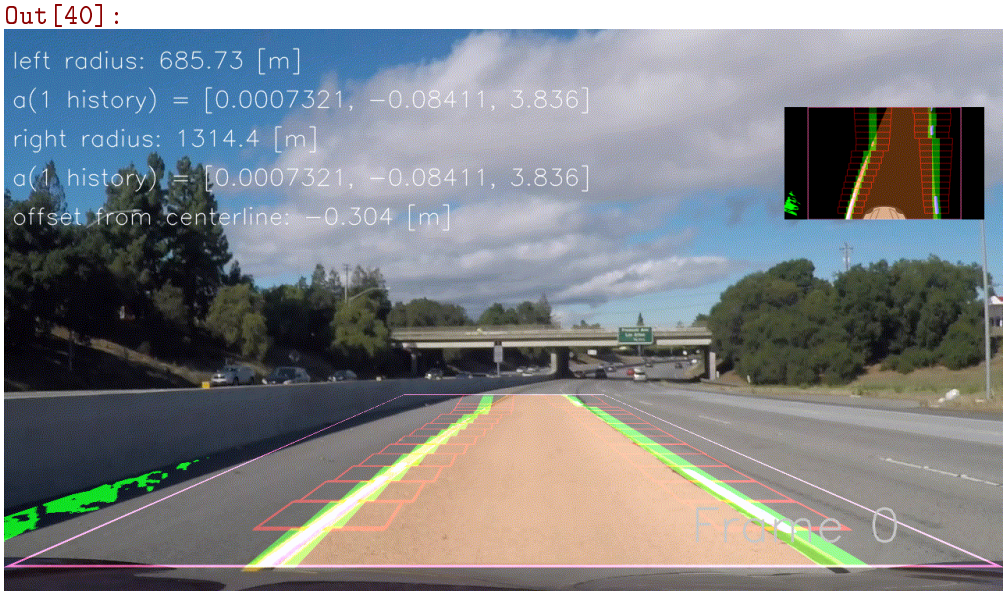
In [39]:
```
gifs['project']
```

Out[39]:



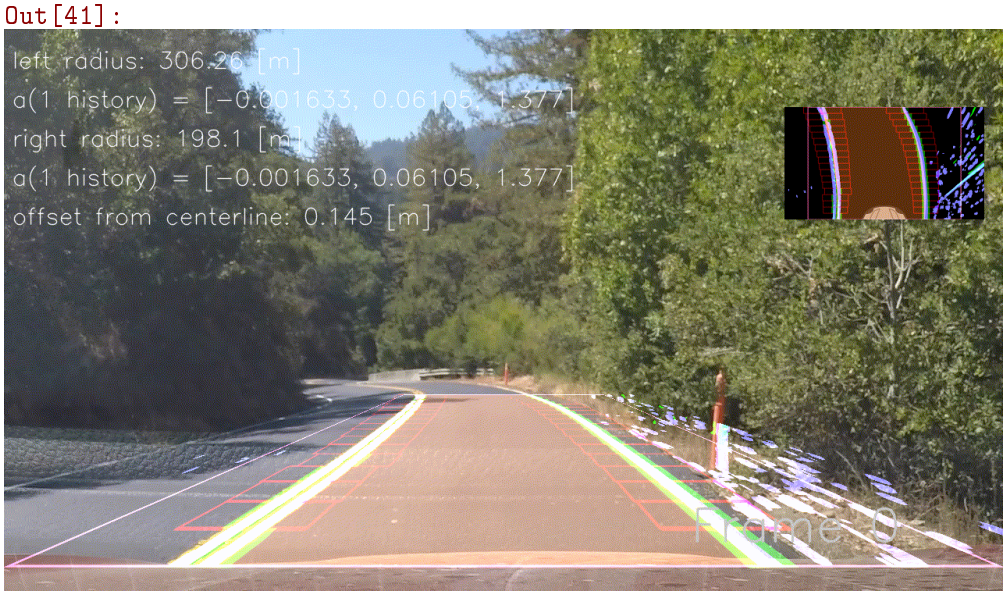
In [40]:
```
gifs['challenge']
```

Out[40]:



```
In [41]: gifs['harder_challenge']
```

Out[41]:



5 Conclusion

The divide-and-conquer approach of the `FullPipeline` is strongly reminiscent of the internal feature-finding strategy of convolutional neural networks (CNNs). For this reason, as well as the extensive per-node tweaking requied to get results out of this method, it's clear that an alternate CNN attempt is warranted.

While results are respectable, there is clear room for improvment, and I can list a few ways in which I might attempt this when I return later.

It would be good to use a hierarchy of pixel-finding strategies. Since this sort of logic was more of the topic of the earlier Hough-transform based assignment, I chose to spend more time on the computer vision aspect of this project.

For simplicity, I simply used the convolutional approach for every frame here. However, using the margin approach (implemented separately in `MarginSearchMarkingFinder`) might both slightly improve speed but also greatly improve stability. Especially e.g. in cases where the broken-white right lane begins with a gap near the camera, using the convolutional approach is prone to sudden jumps in the location of window center in the first layer. (This is especially true of the lower half of the preprocessed images, which tend to contain more noise by virtue of the greater detail of their before-perspective preimages.) The margin-search approach is not prone to this problem. I believe that using the margin-search method in most cases (and reverting to the convolutional method only when a quality metric like number-of-points dips below threshold) would likely fix the remaining small issues in the main project video.

Additionally, it would be good to continue this change-strategy-on-failure technique by simply skipping frames for which the convolutional method also "failed" (per some quality metric). These lane marking fits just would not get integrated into the smoothing.

In this line, it would also be useful to improve the quality metric. I'm fond of the idea of not throwing out low-quality (but not outright wrong) fits completely, using the `WeightedSmoother` that is implemented alongside the currently-used `BoxSmoother`. It seems intuitive that mean

squared error should be included here, but this could be misleading if the number of points found was low (making MSE artificially high). So, some sort of combination is needed.

MSE itself would be defined differently if I used RANSAC instead of normal least-squares `np.polyfit` to find the fitting coefficients--probably I'd need to use MSE within the inliers group defined in RANSAC. Using RANSAC would help greatly, as I'd be able to be less careful with my restrictive masks in the initial OpenCV preprocessing. RANSAC would be able to leave out the many outliers, picking up on the underlying narrow band of dense points.

I would also like to try a predictive model for the evolution of the lane lines. For example, I coul duse a polynomial fit in time of each coefficient, or even just linearly project the last two fit vectors. However, I believe this is to be the topic of a future project, so I'll hold off for now.

Finally, there is a completely different category of error that occurs sometimes in the `harder_challenge_video.mp4` file, where bright sunlight causes reflections of the car dashboard to appear in the windshield. Though not shown here, I did make a small attempt to use (a lack of) optical flow to detect these pixels (see the `DenseOpticalFlow`, `SimplisticOpticalFlow` and `RunningAverage` operations implemented in workers.py). This didn't work, but I belive the intuition is good: temporarily exclude from the preprocessing portions of the video that remain unchanged for several frames.