Annex

Pere Parés Casellas

UOC - Màster en Enginyeria Informàtica

Tesi Final de Màster

David Masip Rodó

30 de desembre de 2014

Índex

1 Consideracions prèvies

En aquest Annex es pot trobar tot el codi que dóna solució als objectius de la tesi així com el que permet retallar exemples positius i negatius per a l'entrenament del classificador *Haar* i el que permet extreure estadístiques d'encert dels vídeos processats a mode de validació. A la Secció 2.4 es poden trobar les instruccions d'ús del programa principal, que poden esser utilitzades com a manual per a l'usuari final.

Juntament amb aquest document es poden trobar, també, tots els arxius font de l'arquitectura així com els dos programes auxiliars i l'arbre de directoris que se sol utilitzar per al seu correcte funcionament. Aquest arbre de directoris pot variar, només cal actualitzar la configuració pertinent:

- */codi*:
  - */codi/assets/*: on es solen dipositar els vídeos o imatges a ser processades
  - */codi/assets/results*: on es solen dipositar els vídeos processats i els arxius *CSV* resultants.
  - */codi/config*: on hi sol haver el fitxer amb els paràmetres de configuració.
  - */codi/haar_cascades*: on hi sol haver els arxius **.xml* resultants de l'entrenament del classificador *Haar* i que s'utilitzen durant el procés de classificació.
  - */codi/src/*: conté tot el codi font de les aplicacions.
- */videos*: vídeos processats dels quals se'n va fent esment durant l'informe.

2 Codi principal

2.1 Mòdul principal

tfm.py:

```
# coding: utf-8

''' Main module - parses the command line parameters, parses the configuration file
    and processes each video while storing the results where needed.
    .. module:: tfm
    .. moduleauthor:: Pere Pares Casellas <ppares.casellas@gmail.com>
'''
```

```
import cv2
import sys, getopt, os, csv, time
import configuration as cfg
import numpy as np
import segmentation as seg
from processing import Processing
from processing import Skeleton
from processing import Segmentation
from processing import HaarLimbClassification
from processing import HaarHeadClassification

CV_CAP_PROP_POS_AVI_RATIO = 2 # Relative position in the video file.
CV_CAP_PROP_FRAME_WIDTH = 3   # Width of the frames in the video stream.
CV_CAP_PROP_FRAME_HEIGHT = 4  # Height of the frames in the video stream.
CV_CAP_PROP_FPS = 5           # Frame rate.
CV_CAP_PROP_FOURCC = 6        # Codec's four character code.
CV_CAP_PROP_FRAME_COUNT = 7   # Number of frames in the video file.

def process_video(origin, destination, segmentation_chain, classification_chain, results_file_name, max_frames=-1):
    ''' Loads the video stream, applies the algorithm chain to all the frames and stores the result.
        :param origin: path to the video that needs to be processed (path + name).
        :param destination: location to store the processed video (path + name).
        :param segmentation_chain: Chain of responsibility object used to apply all the required segmentation algorithms to each frame.
        :type segmentation_chain: Processing
        :param classification_chain: Chain of responsibility object used to apply all the required classification algorithms to each frame.
        :type classification_chain: Processing
        :param results_file_name: file where the results need to be written.
        :param max_frames: maximum number of frames to process.
    '''
    # Load the video to be processed.
    cap = cv2.VideoCapture(origin)

    # Read some important information from the opened stream.
    width = int(cap.get(CV_CAP_PROP_FRAME_WIDTH))
    height = int(cap.get(CV_CAP_PROP_FRAME_HEIGHT))
```

```
frame_rate = cap.get(CV_CAP_PROP_FPS)
n_frames = int(cap.get(CV_CAP_PROP_FRAME_COUNT))
max_frames = n_frames if max_frames == -1 else max_frames

print "Processing " + str(max_frames) + " frames. Please wait ..."

# Define the codec and create the VideoWriter object
fourcc = cv2.cv.CV_FOURCC(*'XVID')
out = cv2.VideoWriter(destination, fourcc, frame_rate, (width, height))

# Prepare the results file.
results_file = open(results_file_name, 'wb')
csv_file = csv.writer(results_file, delimiter=';')
csv_file.writerow(['frame', 'limb1_x', 'limb1_y', 'limb1_roi_x', '
    limb1_roi_y', 'limb1_roi_w', 'limb1_roi_h',
                'limb2_x', 'limb2_y', 'limb2_roi_x', 'limb2_roi_y', '
                    limb2_roi_w', 'limb2_roi_h'])

# Process and store each frame.
n_frames = 1
while cap.isOpened():
    retrn, frame = cap.read()
    if retrn == True:
        print "Processing frame #" + str(n_frames)
        seg_result = segmentation_chain.process(frame)
        cla_result = classification_chain.process(frame)
        result, points = combine_results(frame, seg_result['seg'],
            seg_result['skel'], cla_result['limbs'], cla_result['head'])
        csv_line = [str(n_frames)]
        if len(points) > 0:
            if len(points) == 1:
                csv_line.extend([points[0]['p'][0], points[0]['p'][1],
                    points[0]['r'][0], points[0]['r'][1], points[0]['r'
                    ][2], points[0]['r'][3]])
                csv_line.extend(['', '', '', '', '', ''])
            else:
                csv_line.extend([points[0]['p'][0], points[0]['p'][1],
                    points[0]['r'][0], points[0]['r'][1], points[0]['r'
                    ][2], points[0]['r'][3]])
                csv_line.extend([points[1]['p'][0], points[1]['p'][1],
                    points[1]['r'][0], points[1]['r'][1], points[1]['r'
                    ][2], points[1]['r'][3]])
            else:
                csv_line.extend(['', '', '', '', '', '', '', '', '', '', '', ''])
            csv_file.writerow(csv_line)
            out.write(result if len(result.shape) > 2 else cv2.cvtColor(result, cv2.COLOR_GRAY2BGR))
        else:
            break
        if n_frames > max_frames:
            break
        n_frames = n_frames + 1
    # Release everything once the process is complete.
    cap.release()
    out.release()

def combine_results(original_image, segmented_image, skeleton_image, limb_location, head_location):
    ''' For each found limb's ROI, a mask is created to mark the thicker skeleton where all the processed images have information.
        After that, the algorithm attempts to find the lowest rightmost point of each limb.
        Finally, the mask is applied to a copy of the original image and all the points and ROIs are drawn on top of it.
        :param original_image: original frame as read from the video.
        :param segmented_image: image once segmented.
        :param skeleton_image: skeletonized image.
        :param limb_location: location of the limbs.
        :param head_location: location of the head.
        :returns: final image combining all the information and the points where the limbs were found.
    '''
    result_image = original_image.copy()
    original_image = cv2.cvtColor(original_image, cv2.COLOR_BGR2GRAY)
    segmented_image = cv2.cvtColor(segmented_image, cv2.COLOR_BGR2GRAY)
    skeleton_image = cv2.cvtColor(skeleton_image, cv2.COLOR_BGR2GRAY)

    # 1. Create and apply limb masks.
    limb_mask = np.zeros((original_image.shape[0], original_image.shape[1], 3), dtype=np.uint8)
    points = []
    for (x, y, w, h) in limb_location:
        thicker_skeleton = cv2.dilate(seg.fill_largest_blob(skeleton_image[y:y+
```

```python
            h, x:x+w]), np.ones((3, 3), np.uint8), iterations=2)
        combination = original_image[y:y+h, x:x+w] & segmented_image[y:y+h, x:x+w] & thicker_skeleton
        ret, thresh = cv2.threshold(combination, 1, 255, cv2.THRESH_BINARY)
        point = seg.find_lowest_rightmost_white(thresh, x, y)
        if point == (-1, -1):
            point = (x+w/2, y+h/2)
        points.append({'p':point, 'r':(x, y, w, h)})
        limb_mask[y:y+h, x:x+w, 2] = thresh

    # # 2. Create the head mask.
    # head_mask = np.zeros((original_image.shape[0], original_image.shape[1], 3), dtype = np.uint8)
    # for (x, y, w, h) in head_location:
    #     head_mask[y:y+h, x:x+w, 0] = original_image[y:y+h, x:x+w] & segmented_image[y:y+h, x:x+w]

    # 3. Apply the masks on top of the result image.
    result_image = cv2.addWeighted(result_image, 0.6, limb_mask, 0.9, 0)
    # result_image = cv2.addWeighted(result_image, 0.6, head_mask, 0.5, 0)

    # 4. Draw the points where the limbs have been located on top of the result image.
    for point in points:
        cv2.circle(result_image, point['p'], 3, (0, 255, 0), -1)
        (x, y, w, h) = point['r']
        cv2.rectangle(result_image, (x, y), (x+w, y+h), (0, 255, 0), 1)

    # 5. Return the result.
    return result_image, points

def parse_arguments(argv):
    ''' Displays help or retrieves the configuration file path and results path from the command line.
        :param argv: command line parameters.
        :returns: configuration file path, results path
    '''
    # Default configuration file path:
    config_file_path = '../config/configuration.cfg'
    results_path = '../assets/results/'

    # Parse the command line parameters:
```

```
    try:
        opts, args = getopt.getopt(argv, 'hc:r:', ['config=', 'results='])
    except getopt.GetoptError:
        print 'tfm.py −c <configfile> −r <resultspath>'
        sys.exit()
    for opt, arg in opts:
        if opt == '−h':
            print 'tfm.py −c <configfile> −r <resultspath>'
            sys.exit()
        elif opt in ("−c", "−−config"):
            config_file_path = arg
        elif opt in ("−r", "−−results"):
            results_path = arg

    if not os.path.isfile(config_file_path):
        print 'Error: the specified configuration file does not exist: ' +
            config_file_path
        sys.exit()
    if not os.path.isdir(results_path):
        print 'Error: the specified results path does not exist: ' +
            results_path
        sys.exit()
    return config_file_path, results_path

def main(argv):
    ''' Reads the parameters from the command line, prepares the computer
        vision algorithm's chain
        of responsibility objects and starts the whole process for all the
            videos.
        :param argv: parameters given from the command line.
    '''
    print '——————————————————————————— TFM − Pere Pares Casellas
        ———————————————————————————'


    # Parse the command line arguments, if any:
    config_file_path, results_path = parse_arguments(argv)

    # Parse the configuration file:
    config = cfg.Configuration(config_file_path)
    if not config.read_in():
        sys.exit()

    print config

    # --- 1. Prepare the segmentation chain.
    segmentation_chain = Processing(config)

    # 1.1 Segment each frame.
    segm = Segmentation(config)
    segmentation_chain.set_next(segm)

    # 1.2 Skeletonize the resulting segmentation.
    skel = Skeleton(config)
    segm.set_next(skel)

    # --- 2. Prepare the detection chain.
    classification_chain = Processing(config)

    # 2.1 Apply Haar classification to detect the limbs:
    haar_limbs = HaarLimbClassification(config)
    classification_chain.set_next(haar_limbs)

    # 2.2 Apply Haar classification to detect the head:
    haar_head = HaarHeadClassification(config)
    haar_limbs.set_next(haar_head)

    # --- 3. Process all the videos applying the chain of algorithms to each frame and store the result.
    print '\n- Starting the artificial vision processing of all the videos:\n'
    for index in range(len(config.input_videos)):
        if config.input_videos[index] != '' and config.output_videos[index] != '':
            print 'Processing video ' + config.input_videos[index] + ':'
            results_file_name = results_path + '/results_' + os.path.basename(config.input_videos[index]).split('.')[0] + '_' + \
                                time.strftime('%Y%m%d%H%M%S.csv', time.localtime())
            process_video(config.input_videos[index], config.output_videos[index],
                          segmentation_chain, classification_chain,
                          results_file_name, max_frames=config.n_frames_to_process)


if __name__ == '__main__':
    main(sys.argv[1:])
```

2.2 Mòdul de processat

processing.py:

```
# coding: utf-8

''' Used to define and execute all the computer vision algorithms over all the video's frames.
    .. module:: processing
    .. moduleauthor:: Pere Pares Casellas <ppares.casellas@gmail.com>
'''

import cv2
import segmentation as segm
import skeletonization as skel

class Processing(object):
    ''' Implementation of the software engineering chain of responsibility pattern. Used to define
        and execute the proper computer vision algorithms over all the frames of the given video in a specific order.
        :ivar config: Application's configuration manager.
        :type config: Configuration
        :ivar next_: Reference to the next algorithm in the chain that needs to be applied.
        :type next_: Processing
    '''

    config = None
    next_ = None

    def __init__(self, config):
        self.config = config

    def set_next(self, next_processing):
        ''' Sets the next algorithm to be applied after the current one.
            :param next_processing: next algorithm to be applied.
            :type next_processing: Processing
        '''
        self.next_ = next_processing

    def process(self, original):
        ''' Applies all the algorithms in the chain and returns their result.
            :param original: frame to be processed.
            :returns: processed frame.
        '''
        if self.next_ is not None:
            return self.next_.process(original)
        return original

class Skeleton(Processing):
    ''' Skeletonizes the given image and passes the result to the next algorithm in the chain.
        Implements a single step of the chain of responsibility process.
    '''

    def __init__(self, config):
        Processing.__init__(self, config)

    def process(self, original):
        ''' Skeletonizes the given image, passes the result to the next algorithm - if any - and returns the final result.
            :param original: frame to be processed.
            :returns: processed frame.
        '''
        print '\tSkeletonizing ...'
        # 1. Original image to gray level - if required:
        if len(original.shape) > 2:
            gray = cv2.cvtColor(original, cv2.COLOR_BGR2GRAY)
        else:
            gray = original.copy()
        # 2. Skeleton
        if self.config.skel_algorithm == 0:
            skeleton = cv2.cvtColor(skel.skeletonize(gray), cv2.COLOR_GRAY2BGR)
        elif self.config.skel_algorithm == 1:
            skeleton = cv2.cvtColor(skel.zhang_suen(gray), cv2.COLOR_GRAY2BGR)
        else:
            skeleton = cv2.cvtColor(skel.guo_hall(gray), cv2.COLOR_GRAY2BGR)

        if self.next_ is not None:
            res = self.next_.process(skeleton)
            res['skel'] = skeleton
            return res
        return {'skel': skeleton}

class Segmentation(Processing):
    ''' Segments the given image and passes the result to the next algorithm in the chain.
        Implements a single step of the chain of responsibility process.
    '''

    def __init__(self, config):
        Processing.__init__(self, config)

    def process(self, original):
        ''' Segments the given image using grabcut, passes the result to the next algorithm - if any - and returns the final result.
            :param original: frame to be processed.
            :returns: processed image.
        '''
        print '\tSegmenting ...'
        # Copy the original image
        image = original.copy()

        #1. Original image to gray level - if required:
        if len(image.shape) > 2:
            gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
        else:
            gray = image

        #2. Apply Otsu + Gaussian binarization:
        blur = cv2.GaussianBlur(gray, (5, 5), 0)
        ret, thresh = cv2.threshold(blur, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)

        #3. Find the largest blob:
        contours, hierarchy = cv2.findContours(thresh, 1, 2)
        area = 0
        largest_contour = None
        for contour in contours:
            if cv2.contourArea(contour) > area:
                largest_contour = contour

        #4. Get the largest contour's bounding box
```

```python
        if largest_contour is not None:
            x, y, w, h = cv2.boundingRect(largest_contour)
            #cv2.rectangle(image, (x, y), (x+w, y+h+lower_offset), (0, 255, 0), 1)
        else:
            print 'Error: no contours found within the given image.'

        #5. Apply grab-cut:
        grabcut, truefg, truebg, mask = segm.grab_cut(image, (x, y, w, h + self.config.bounding_box_lower_offset),
                                                       self.config.grabcut_iterations, self.config.circle_x,
                                                       self.config.circle_y, self.config.circle_radius)
        if self.next_ is not None:
            seg = self.next_.process(grabcut)
            seg['seg'] = grabcut
            return seg
        return {'seg' : grabcut}

class HaarLimbClassification(Processing):
    ''' Tries to detect the mice's limbs using the trained Haar limb classifier and passes the result to the next algorithm in the chain.
        Implements a single step of the chain of responsibility process.
    '''

    def __init__(self, config):
        Processing.__init__(self, config)

    def process(self, original):
        ''' Attempts to detect the mice's limbs, passes the result to the next algorithm - if any - and returns the final result.
            :param original: frame to be processed.
            :returns: processed image.
        '''
        print '\tDetecting the limbs ...'

        # Copy the original image:
        image = original.copy()
        #1. Original image to gray level - if required:
        if len(image.shape) > 2:
            gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
        else:
            gray = image

        #2. Load the limb cascade:
        limb_cascade = cv2.CascadeClassifier(self.config.haar_limb_cascade)

        #3. Detect the limbs and draw a bounding box around them:
        limbs = limb_cascade.detectMultiScale(gray, 1.3, 5)
        for (x, y, w, h) in limbs:
            cv2.rectangle(image, (x, y), (x+w, y+h), (255, 0, 0), 2)

        if self.next_ is not None:
            limb = self.next_.process(image)
            limb['limb_img'] = image
            limb['limbs'] = limbs
            return limb
        return {'limb_img' : image, 'limbs' : limbs}

class HaarHeadClassification(Processing):
    ''' Tries to detect the mice's head using the trained Haar head classifier and passes the result to the next algorithm in the chain.
        Implements a single step of the chain of responsibility process.
    '''

    def __init__(self, config):
        Processing.__init__(self, config)

    def process(self, original):
        ''' Attempts to detect the mice's head, passes the result to the next algorithm - if any - and returns the final result.
            :param original: frame to be processed.
            :returns: processed image.
        '''
        print '\tDetecting the head ...'

        # Copy the original image:
        image = original.copy()

        #1. Original image to gray level - if required:
```

```
        if len(image.shape) > 2:
            gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
        else:
            gray = image

        #2. Load the head cascade:
        head_cascade = cv2.CascadeClassifier(self.config.haar_head_cascade)

        #3. Detect the head and draw a bounding box around it:
        head = head_cascade.detectMultiScale(gray, 1.3, 5)
        for (x, y, w, h) in head:
            cv2.rectangle(image, (x, y), (x+w, y+h), (0, 255, 0), 2)

        if self.next_ is not None:
            head = self.next_.process(image)
            head['head_img'] = image
            head['head'] = head
            return head
        return {'head_img' : image, 'head' : head}
```

2.2.1 Mòdul de segmentació

segmentation.py:

```
# coding: utf-8

''' Algorithms used to segment images.
    .. module:: segmentation
    .. moduleauthor:: Pere Pares Casellas <ppares.casellas@gmail.com>
'''

import numpy as np
import cv2

def kmeans(img, K):
    ''' Applies the k-means algorithm over the given image using K clusters
        until the finishing criteria is met and returns the result.
        :param img: image to be segmented.
        :param K: number of centroids to be used.
        :returns: segmented image.
    '''
    Z = img.reshape((-1, 1))

    # convert to np.float32
    Z = np.float32(Z)

    # define criteria, number of clusters(K) and apply kmeans()
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
    ret, label, center = cv2.kmeans(Z, K, criteria, 10, cv2.
        KMEANS_RANDOM_CENTERS)

    # Now convert back into uint8, and make original image
    center = np.uint8(center)
    res = center[label.flatten()]
    res2 = res.reshape((img.shape))
    return res2

def find_lowest_rightmost_white(binary_img, offset_x, offset_y):
    ''' Looks for the lowest rightmost value in a binary image.
        :param binay_img: binary image.
        :param offset_x: offset to apply to the x of the found point.
        :param offset_y: offset to apply to the y of the found point.
    '''
    point = (-1, -1)
    for i in range(binary_img.shape[0]):
        for j in range(binary_img.shape[1]):
            if binary_img.item(i, j) > 0:
                point = (j+offset_x, i+offset_y)
    return point

def fill_largest_blob(original):
    ''' Binarizes the passed image using Otsu's method and returns the
        resulting binary
        image containing only the largest blob found.
        :param original: image to be processed.
        :returns: binary image containing only the largest blob found.
    '''
    # Copy the original image
    image = original.copy()

    #1. Original image to gray level - if required:
    if len(image.shape) > 2:
        gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    else:
        gray = image

    blur = cv2.GaussianBlur(gray, (5, 5), 0)
    ret, thresh = cv2.threshold(blur, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
    binary = thresh.copy()

    contours, hierarchy = cv2.findContours(thresh, 1, 2)

    area = 0
    largest_contour = None
    for contour in contours:
        if cv2.contourArea(contour) > area:
            largest_contour = contour

    if largest_contour is not None:
        # Remove all the other blobs from the binary image
        blob = np.zeros(binary.shape[:2], np.uint8)
        cv2.drawContours(blob, [largest_contour], 0, (255), -1)
        return blob
    return np.zeros(binary.shape[:2], np.uint8)

def grab_cut(img, rect, iterations, circle_x, circle_y, circle_radius):
    ''' Applies the grabcut algorithm to segment the passed image and returns the result.
        :param img: image to be processed. @pre: BGR required.
        :param rect: rectangle used to perform the first grabcut step - must contain the object to segment.
        :param iterations: number of iterations for each grabcut step.
        :param circle_x: central point's x component for the true background circle (to properly segment the wheel).
        :param circle_y: central point's y component for the true background circle (to properly segment the wheel).
        :param circle_radius: radius of the circle used to describe the true background to properly segment the wheel.
        :returns: segmented image.
        :returns: possible foreground mask.
        :returns: true background mask.
        :returns: combined mask used to perform the grabcut steps.
    '''
    img2 = img.copy()

    # 1. Initialise all the mask to possible background
    mask = np.zeros(img.shape[:2], np.uint8) + cv2.GC_PR_BGD

    # 2. Execute grab cut using only the initial rectangle information
    background_model = np.zeros((1, 65), np.float64)
    foreground_model = np.zeros((1, 65), np.float64)
    cv2.grabCut(img2, mask, rect, background_model, foreground_model, 1, cv2.GC_INIT_WITH_RECT)

    # 3. Modify the mask using true background and true background information
    # 3.1 Fill the largest blob - this information will be used as possible foreground.
    possiblefg = fill_largest_blob(img)
    # 3.2 Fill the lower circle - this information will be used as true background.
    truebg = np.zeros(img.shape[:2], np.uint8)
    cv2.circle(truebg, (circle_x, circle_y), circle_radius, (255), -1)

    # 3.3 Update the mask using possible and true background information
    for i in xrange(truebg.shape[0]):
        for j in xrange(truebg.shape[1]):
            if possiblefg.item(i, j) == 0:
                mask.itemset((i, j), cv2.GC_PR_BGD)
            if truebg.item(i, j) == 255:
                mask.itemset((i, j), cv2.GC_BGD)
            elif possiblefg.item(i, j) == 255:
                mask.itemset((i, j), cv2.GC_PR_FGD)

    # Save the mask before applying grab-cut to display it later, if required
    mask_c = mask.copy()
    mask_c[mask == cv2.GC_BGD] = 0      # true background is black
    mask_c[mask == cv2.GC_PR_BGD] = 63  # possible background is dark grey
    mask_c[mask == cv2.GC_FGD] = 255     # foreground is white
    mask_c[mask == cv2.GC_PR_FGD] = 192 # possible foreground is light grey

    # 4. Execute grab cut on the image using the updated mask with possible foreground and true background
    background_model = np.zeros((1, 65), np.float64)
    foreground_model = np.zeros((1, 65), np.float64)
    cv2.grabCut(img2, mask, rect, background_model, foreground_model, iterations, cv2.GC_INIT_WITH_MASK)
```

```python
    # 5. Use the obtained mask to get the final segmented image.
    mask = np.where((mask == 2) | (mask == 0), 0, 1).astype('uint8')
    output = img2*mask[:, :, np.newaxis]

    return output, possiblefg, truebg, mask_c
```

2.2.2 Mòdul d'esqueletització

skeletonization.py:

```python
# coding: utf-8

''' Algorithms used to segment images.
    .. module:: skeletonization
    .. moduleauthor:: Pere Pares Casellas <ppares.casellas@gmail.com>
'''

import cv2
import scipy.ndimage.morphology as morph
import numpy as np

def skeletonize(img):
    ''' Skeletonizes the gray level image passed by parameter using scipy's morphology module.
        :param img: image to be skeletonized. @pre: gray level required.
        :returns: skeletonized image.
    '''
    # Gaussian to smooth the image and remove noise
    gray = cv2.GaussianBlur(img, (3, 3), 0)
    # Inverted Otsu binarization
    ret, thresh = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)
    element = cv2.getStructuringElement(cv2.MORPH_CROSS, (3, 3))
    thresh = 255 - thresh
    thresh = cv2.dilate(thresh, element, iterations=3)
    h1 = np.array([[0, 0, 0], [0, 1, 0], [1, 1, 1]])
    m1 = np.array([[1, 1, 1], [0, 0, 0], [0, 0, 0]])
    h2 = np.array([[0, 0, 0], [1, 1, 0], [0, 1, 0]])
    m2 = np.array([[0, 1, 1], [0, 0, 1], [0, 0, 0]])
    hit_list = []
    miss_list = []
    for k in range(4):
        hit_list.append(np.rot90(h1, k))
        hit_list.append(np.rot90(h2, k))
        miss_list.append(np.rot90(m1, k))
        miss_list.append(np.rot90(m2, k))
    thresh = thresh.copy()
    while True:
        last = thresh
        for hit, miss in zip(hit_list, miss_list):
            hm = morph.binary_hit_or_miss(thresh, hit, miss)
            thresh = np.logical_and(thresh, np.logical_not(hm))
        if np.all(thresh == last):
            break
    return thresh.astype(np.uint8)*255

def __zhang_suen_iteration(img, iteration):
    ''' Performs one thinning iteration for the Zhang-Suen's thinning algorithm.
        :param img: image to perform the thinning iteration to.
        :param iteration: 0 or 1.
        :returns: result of applying one thinning iteration to the given image.
    '''
    marker = np.zeros((img.shape[0], img.shape[1]), np.uint8)
    for i in range(1, img.shape[0]-1):
        for j in range(1, img.shape[1]-1):
            p2 = img.item(i-1, j)
            p3 = img.item(i-1, j+1)
            p4 = img.item(i, j+1)
            p5 = img.item(i+1, j+1)
            p6 = img.item(i+1, j)
            p7 = img.item(i+1, j-1)
            p8 = img.item(i, j-1)
            p9 = img.item(i-1, j-1)

            A = (p2 == 0 and p3 == 1) + (p3 == 0 and p4 == 1) + \
                (p4 == 0 and p5 == 1) + (p5 == 0 and p6 == 1) + \
                (p6 == 0 and p7 == 1) + (p7 == 0 and p8 == 1) + \
                (p8 == 0 and p9 == 1) + (p9 == 0 and p2 == 1)

            B = p2 + p3 + p4 + p5 + p6 + p7 + p8 + p9
            m1 = (p2 * p4 * p6) if iteration == 0 else (p2 * p4 * p8)
            m2 = (p4 * p6 * p8) if iteration == 0 else (p2 * p6 * p8)
```

```python
            if A == 1 and (B >= 2 and B <= 6) and m1 == 0 and m2 == 0:
                marker.itemset((i, j), 1)
    return np.logical_and(img, np.logical_not(marker)).astype('uint8')

def zhang_suen(img):
    ''' Applies the Zhang-Suen thinning algorithm to the given gray level image.
        :param img: Image to skeletonize. @pre: gray level image.
        :returns: Skeletonized image.
    '''
    # Gaussian to smooth the image and remove noise
    gray = cv2.GaussianBlur(img, (3, 3), 0)
    # Inverted Otsu binarization
    ret, thresh = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
    thresh = thresh / 255
    prev = np.zeros((thresh.shape[0], thresh.shape[1]), np.uint8)
    while True:
        thresh = __zhang_suen_iteration(thresh, 0)
        thresh = __zhang_suen_iteration(thresh, 1)
        diff = cv2.absdiff(thresh, prev)
        prev = thresh.copy()
        if cv2.countNonZero(diff) <= 0:
            break
    return thresh * 255

def __guo_hall_iteration(img, iteration):
    ''' Performs one thinning iteration for the Guo-Hall's thinning algorithm.
        :param img: image to perform the thinning iteration to.
        :param iteration: 0 or 1.
        :returns: result of applying one thinning iteration to the given image.
    '''
    marker = np.zeros((img.shape[0], img.shape[1]), np.uint8)
    for i in range(1, img.shape[0]-1):
        for j in range(1, img.shape[1]-1):
            p2 = img.item(i-1, j)
            p3 = img.item(i-1, j+1)
            p4 = img.item(i, j+1)
            p5 = img.item(i+1, j+1)
            p6 = img.item(i+1, j)
            p7 = img.item(i+1, j-1)
            p8 = img.item(i, j-1)
            p9 = img.item(i-1, j-1)

            C = (~p2 & (p3 | p4)) + (~p4 & (p5 | p6)) + (~p6 & (p7 | p8)) + (~p8 & (p9 | p2))
            N1 = (p9 | p2) + (p3 | p4) + (p5 | p6) + (p7 | p8)
            N2 = (p2 | p3) + (p4 | p5) + (p6 | p7) + (p8 | p9)
            N = N1 if N1 < N2 else N2
            m = ((p6 | p7 | ~p9) & p8) if iteration == 0 else ((p2 | p3 | ~p5) & p4)

            if (C == 1 and (N >= 2 and N <= 3)) & (m == 0):
                marker.itemset((i, j), 1)

    return np.logical_and(img, np.logical_not(marker)).astype('uint8')

def guo_hall(img):
    ''' Applies the Guo-Hall thinning algorithm to the given gray level image.
        :param img: Image to skeletonize. @pre: gray level image.
        :returns: Skeletonized image.
    '''
    # Gaussian to smooth the image and remove noise
    gray = cv2.GaussianBlur(img, (3, 3), 0)
    # Inverted Otsu binarization
    ret, thresh = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
    thresh = thresh / 255
    prev = np.zeros((thresh.shape[0], thresh.shape[1]), np.uint8)
    while True:
        thresh = __guo_hall_iteration(thresh, 0)
        thresh = __guo_hall_iteration(thresh, 1)
        diff = cv2.absdiff(thresh, prev)
        prev = thresh.copy()
        if cv2.countNonZero(diff) <= 0:
            break
    return thresh * 255
```

2.3 Gestor de configuració

Codi del gestor de configuració (*configuration.py*):

```python
# coding: utf-8
```

```
''' Used to read and provide configuration parameters to the main application.
    .. module:: configuration
    .. moduleauthor:: Pere Pares Casellas <ppares.casellas@gmail.com>
'''

import ConfigParser

class Configuration(object):
    ''' Configuration class used to read and provide parameters to the main application.
        :ivar __config_file_path: path to the configuration file.
        :ivar bounding_box_lower_offset: offset to be applied under the grabcut's initial bounding box.
        :ivar grabcut_iterations: number of grabcut iterations for each frame.
        :ivar skel_algorithm: algorithm to be used to compute the skeleton.
        :ivar n_frames_to_process: number of frames to process for the given video.
        :ivar input_videos: video/s to process.
        :ivar output_videos: processed video/s.
        :ivar haar_limb_cascade: Haar limb cascade classifier file path.
        :ivar haar_head_cascade: Haar head cascade classifier file path.
        :ivar circle_x: central point's x component for the true background circle used for grabcut.
        :ivar circle_y: central point's y component for the true background circle used for grabcut.
        :ivar circle_radius: radius of the circle used to describe grabcut's true background.
    '''

    __config_file_path = ''

    bounding_box_lower_offset = 0
    grabcut_iterations = 0
    skel_algorithm = ''
    n_frames_to_process = 0
    input_videos = []
    output_videos = []
    haar_limb_cascade = ''
    haar_head_cascade = ''
    circle_x = 0
    circle_y = 0
    circle_radius = 0

    def __init__(self, file_path):
        self.__config_file_path = file_path

    def read_in(self):
        ''' Reads the configuration from the given configuration file.
            :returns: whether the parsing was successful.
        '''
        print 'Reading configuration from: ' + self.__config_file_path
        try:
            conf = ConfigParser.RawConfigParser()
            conf.read(self.__config_file_path)

            # segmentation:
            self.bounding_box_lower_offset = conf.getint('segmentation', '
                bounding_box_lower_offset')
            self.grabcut_iterations = conf.getint('segmentation', '
                grabcut_iterations')

            # skeleton:
            self.skel_algorithm = conf.getint('skeleton', 'algorithm')

            # video:
            self.n_frames_to_process = conf.getint('video', '
                n_frames_to_process')
            self.input_videos = [x.strip() for x in conf.get('video', '
                input_videos').split(',')]
            self.output_videos = [x.strip() for x in conf.get('video', '
                output_videos').split(',')]
            if len(self.input_videos) != len(self.output_videos):
                print 'input_videos and output_videos lists should be of the
                    same size. Check the configuration file.'
                return False

            # classification:
            self.haar_limb_cascade = conf.get('classification', '
                haar_limb_cascade').strip()
            self.haar_head_cascade = conf.get('classification', '
                haar_head_cascade').strip()


            # constants:
            self.circle_x = conf.getint('constants', 'circle_x')
            self.circle_y = conf.getint('constants', 'circle_y')
            self.circle_radius = conf.getint('constants', 'circle_radius')

        except ConfigParser.Error as e:
            print 'Exception thrown while reading the configuration file: ' + e
            return False
        return True

    def __str__(self):
        ''' Stringifies the current object's attributes.
            :returns: string representation of the configuration parameters.
        '''
        return 'Configuration: ' + str({'bounding_box_lower_offset': self.bounding_box_lower_offset,
                                        'grabcut_iterations': self.grabcut_iterations,
                                        'skel_algorithm': self.skel_algorithm,
                                        'n_frames_to_process': self.n_frames_to_process,
                                        'input_videos': self.input_videos,
                                        'output_videos': self.output_videos,
                                        'haar_limb_cascade': self.haar_limb_cascade,
                                        'haar_head_cascade': self.haar_head_cascade,
                                        'circle_x': self.circle_x,
                                        'circle_y': self.circle_y,
                                        'circle_radius': self.circle_radius})
```

Fitxer de configuració (*configuration.cfg*):

```
##
# Configuration parameters:
# - segmentation:
#   + bounding_box_lower_offset [int]: lower offset for the grabcut's initial bounding box (in pixels).
#   + grabcut_iterations [int]: number of grabcut's iterations for each frame.
# - skeleton:
#   + algorithm [0: scipy, 1: zhang-suen, 2: guo-hall]
# - video:
#   + n_frames_to_process [-1, n]: use -1 to process all the available frames.
#   + input_videos: input videos' path.
#   + output_videos: output videos' path.
# - classification:
#   + haar_limb_cascade: Haar limb cascade classifier file path.
#   + haar_head_cascade: Haar head cascade classifier file path.
# - constants:
#   + circle_x: central point's x component for the true background circle used to properly segment the wheel.
#   + circle_y: central point's y component for the true background circle used to properly segment the wheel.
#   + circle_radius: radius of the circle used to describe the true background to properly segment the wheel.
##

[segmentation]
bounding_box_lower_offset: 20
grabcut_iterations: 3

[skeleton]
algorithm: 0

[video]
n_frames_to_process: -1
input_videos: ../assets/PS3_Vid32.avi, ../assets/PS3_Vid36.avi, ../assets/PS3_Vid44.avi, ../assets/PS3_Vid64.avi, ../assets/PS3_Vid81.avi
output_videos: ../assets/results/output_Vid32.avi, ../assets/results/output_Vid36.avi, ../assets/results/output_Vid44.avi, ../assets/results/output_Vid64.avi, ../assets/results/output_Vid81.avi

[classification]
haar_limb_cascade: ../haar_cascades/limbs_cascade.xml
haar_head_cascade: ../haar_cascades/head_cascade.xml

[constants]
circle_x: 170
circle_y: 380
circle_radius: 200
```

2.4 Instruccions d'ús

En aquest apartat es duu a terme una descripció de les instruccions d'ús del codi principal desenvolupat per a donar solució als objectius marcats per aquesta tesi de màster.

Per a desplegar l'eina només fa falta copiar l'arbre de directoris amb el codi pertinent a una màquina que ja disposi dels requisits mínims (*Python 2.7.3*, *OpenCV 2.4.9*, *NumPy*, *SciPy*). A continuació es mostra l'arbre de directoris típic per poder executar l'aplicació:

- */assets/*: on es solen dipositar els vídeos o imatges a ser processades. El fitxer de configuració permet especificar qualsevol altre *path*.
- */assets/results*: on es solen dipositar els vídeos processats i els arxius *CSV* resultants. El fitxer de configuració permet especificar qualsevol altre *path*.
- */config*: on hi sol haver el fitxer amb els paràmetres de configuració. A l'hora de cridar l'aplicació per línia de comandes, es pot especificar la localització del fitxer de configuració sigui on sigui.
- */haar_cascades*: on hi sol haver els arxius **.xml* resultants de l'entrenament del classificador *Haar* i que s'utilitzen durant el procés de classificació. El fitxer de configuració permet especificar qualsevol altre *path*.
- */src/*: conté tot el codi font de les aplicacions.

Així doncs, un cop es disposa de l'aplicació desplegada, cal editar el fitxer de configuració per indicar a l'aplicació on pot trobar la informació que requereix i quins paràmetres ha d'utilitzar per realitzar certes accions. El fitxer de configuració es pot consultar a la Secció 2.3 o directament al codi font entregat juntament amb aquest document. Al propi fitxer de configuració ja es descriu què fa cada paràmetre però, de totes maneres, seguidament s'explicarà per a què serveix cada un d'ells:

- *segmentation*: paràmetres relacionats amb el procés de segmentació.
  - *bounding_box_lower_offset*: *offset* en píxels que cal aplicar a la part inferior del *bounding box* trobat automàticament per poder realitzar el primer pas de *Grab-cut*. Es permet especificar per seguretat, per no perdre cap extremitat abans de començar a aplicar el procediment de segmentació.
    - *grabcut_iterations*: nombre d'iteracions *Grab-cut* per cada fotograma.
- *skeleton*: permet especificar l'algorisme a utilitzar per a realitzar el procediment de construcció de l'esquelet topològic. Hi ha tres possibilitats, totes elles explicades al document principal:
  - *0*: Algorisme implementat utilitzant funcions de la llibreria *SciPy* (més ràpid que la resta).

  - *1*: Algorisme *Zhang-Suen* implementat íntegrament en *Python* (lent degut a això).
  - *2*: Algorisme *Guo-Hall* implementat íntegrament en *Python* (lent degut a això).

- *video*: paràmetres relacionats amb els vídeos a processar.
  - *n_frames_to_process*: nombre de fotogrames a processar. En cas de ser "-1", es processaran tots els fotogrames del vídeo.
  - *input_videos*: llista separada per comes dels vídeos que cal processar.
  - *output_videos*: llista separada per comes dels noms dels vídeos resultants de processar els de la llista anterior. Ha d'existir un nom de sortida per cada un d'entrada.
- *classification*: paràmetres relacionats amb els classificadors *Haar* entrenats.
  - *haar_limb_cascade*: resultat de l'entrenament del classificador d'extremitats. Requerit pel procés de classificació.
  - *haar_head_cascade*: resultat de l'entrenament del classificador de caps. Requerit pel procés de classificació.
- *constants*: constants utilitzades durant tot el processat del vídeo; relacionades amb la segmentació.
  - *circle_x*: coordenada x que defineix el punt central del cercle utilitzat per a marcar com a *true background* la roda on camina el ratolí.
  - *circle_y*: coordenada y que defineix el punt central del cercle utilitzat per a marcar com a *true background* la roda on camina el ratolí.
  - *circle_radius*: radi del cercle utilitzat per a marcar com a *true background* la roda on camina el ratolí.

Un cop adaptats els paràmetres, l'aplicació ja es pot cridar des de línia de comandes:

```
python tfm.py -c <configfile> -r <resultspath>
```

Essent els paràmetres *-c* i *-r* opcionals:

- *-c*: fitxer de configuració, *path* inclòs (*../config/configuration.cfg* per defecte).
- *-r*: *path* on cal guardar els resultats (*../assets/results/* per defecte).

Durant l'execució, l'aplicació anirà informant del que realitza per cada fotograma fins que els hagi processat tots per cada vídeo que faci falta:

```
———————————————————— TFM – Pere Pares Casellas ————————————————————
Reading configuration from: ../config/configuration.cfg
Configuration: {'circle_radius': 200, 'circle_y': 380, 'grabcut_iterations': 3, 'skel_algorithm': 0, 'haar_limb_cascade': '../haar_cascades/limbs_cascade.xml', 'n_frames_to_process': -1, 'circle_x': 170, 'bounding_box_lower_offset': 20, 'haar_head_cascade': '../haar_cascades/head_cascade.xml', 'output_videos': ['../assets/results/final_Vid36_slow_framerate.avi'], 'input_videos': ['../assets/PS3_Vid36.avi']}

- Starting the artificial vision processing of all the videos:

Processing video ../assets/PS3_Vid36.avi:
Processing 479 frames. Please wait ...
Processing frame #1
  Segmenting ...
  Skeletonizing ...
  Detecting the limbs ...
  Detecting the head ...
...
```

3 Codi per captar mostres pel classificador *Haar*

haar_sampler.py:

```python
# coding: utf-8

''' Used to gather data for the Haar classifier training
    .. module:: haar_sampler
    .. moduleauthor:: Pere Pares Casellas <ppares.casellas@gmail.com>
'''

import cv2
import sys, getopt, os
import numpy as np

BLUE = [255, 0, 0]
rect = (0, 0, 1, 1)
img = None
img2 = None
rectangle = False

def on_mouse(event, x, y, flags, param):
    ''' Callback for the mouse event - used to draw the ROIs '''
    global img, img2, rectangle, rect, ix, iy

    # Draw Rectangle
    if event == cv2.EVENT_LBUTTONDOWN:
        rectangle = True
        ix, iy = x, y
    elif event == cv2.EVENT_MOUSEMOVE:
        if rectangle == True:
            img = img2.copy()
            cv2.rectangle(img, (ix, iy), (x, y), BLUE, 2)
            rect = (min(ix, x), min(iy, y), max(ix, x), max(iy, y))
    elif event == cv2.EVENT_LBUTTONUP:
        rectangle = False
        cv2.rectangle(img, (ix, iy), (x, y), BLUE, 2)
        rect = (min(ix, x), min(iy, y), max(ix, x), max(iy, y))

def main(argv):
    ''' Loads the required video and lets the user gather data from each frame of it. '''
    global img, img2, rectangle, rect

    video_file_path = '../assets/PS3_Vid81.avi'
    storing_path = '../assets/haar_training'
    positives = ''
    negatives = ''
    positive_example = True
    positive_identifier = 0
    negative_identifier = 0

    try:
        opts, args = getopt.getopt(argv, 'hv:p:t:f:', ['video=', 'path='])
    except getopt.GetoptError:
        print 'tfm.py -v <videofile>'
        sys.exit()
    for opt, arg in opts:
        if opt == '-h':
            print 'haar_sampler.py -v <videofile> -p <storing_path> -t <true_identifier> -f <false_identifier>'
            sys.exit()
        elif opt in ('-v', '--video'):
            video_file_path = arg
        elif opt in ('-p', '--path'):
            storing_path = arg
        elif opt in ('-t', '--true_identifier'):
                positive_identifier = int(arg)
            elif opt in ('-f', '--false_identifier'):
                negative_identifier = int(arg)

    if not os.path.isfile(video_file_path):
        print 'Error - the specified video does not exist: ' + video_file_path
        sys.exit()
    if not os.path.isdir(storing_path):
        print 'Error - the specified directory does not exist: ' + storing_path
        sys.exit()
    else:
        positives = storing_path + '/positives/'
        negatives = storing_path + '/negatives/'
        if not os.path.isdir(positives):
            os.makedirs(positives)
        if not os.path.isdir(negatives):
            os.makedirs(negatives)

    print '------------------------------------------------------------ TFM - Pere Pares Casellas ------------------------------------------------------------'

    try:
        # Open both output files.
        pos_file = open(storing_path + '/info.dat', 'a')
        neg_file = open(storing_path + '/bg.txt', 'a')

        # input window
        cv2.namedWindow('input')
        cv2.setMouseCallback('input', on_mouse)

        print ' Instructions: \n'
        print ' Draw a rectangle around the desired image and:\n - press "t" to mark true examples.\n - press "f" to mark false examples.\n - press "s" to capture and store the ROI.\n - press "n" to go to the next frame.\n'

        # Load the video to be processed.
        stop = False
        cap = cv2.VideoCapture(video_file_path)
        while cap.isOpened():
            retrn, img = cap.read()
            img2 = img.copy()
            if retrn == True:
                while True:
                    cv2.imshow('input', img)
                    k = 0xFF & cv2.waitKey(1)
                    # Key bindings
                    if k == 27:         # esc to exit
                        stop = True
                        break
                    elif k == ord('t'): # Positives
                        print 'Storing ROIs as positive examples.'
                        positive_example = True
                    elif k == ord('f'): # Negatives
                        print 'Storing ROIs as negative examples.'
                        positive_example = False
                    elif k == ord('s'): # Save image
                        a = rect[1] if rect[1] > 0 else 0
                        b = rect[0] if rect[0] > 0 else 0
                        c = rect[3] if rect[3] < img2.shape[0] else img2.shape[0]
                        d = rect[2] if rect[2] < img2.shape[1] else img2.shape[1]
                        roi = img2[a:c, b:d]
                        gray = cv2.cvtColor(roi, cv2.COLOR_BGR2GRAY)
                        if positive_example:
                            cv2.imwrite(positives + str(positive_identifier) + '.png', gray)
                            pos_file.write('positives/' + str(positive_identifier) + '.png 1 0 0 ' + str(gray.shape[1]) + ' ' + str(gray.shape[0]) + '\n')
                            print 'Result saved as a positive example: ' + positives + str(positive_identifier) + '.png'
                            positive_identifier += 1
                        else:
                            cv2.imwrite(negatives + str(negative_identifier) + '.png', gray)
                            neg_file.write('negatives/' + str(negative_identifier) + '.png' + '\n')
                            print 'Result saved as a negative example: ' + negatives + str(negative_identifier) + '.png'
                            negative_identifier += 1
                    elif k == ord('n'): # Next frame
```

```
                        break
            else:
                break
            if stop:
                break
    except Exception as e:
        print 'Exception thrown: ', e
    finally:
        # Release everything once the process is complete.
        cv2.destroyAllWindows()
        cap.release()
        pos_file.close()
        neg_file.close()

if __name__ == '__main__':
    main(sys.argv[1:])
```

4 Codi per validar els resultats a partir de vídeos processats

results_validator.py:

```
# coding: utf-8

''' Module used to retrieve classification statistics out of processed videos by user interaction.
    .. module:: tfm
    .. moduleauthor:: Pere Pares Casellas <ppares.casellas@gmail.com>
'''

import cv2
import sys, getopt, os, time, csv

CV_CAP_PROP_FRAME_COUNT = 7    # Number of frames in the video file.

def parse_arguments(argv):
    ''' Parses the arguments given by command line.
        :param argv: command line parameters.
        :returns: video file path and results path.
    '''
    video_file_path = '../assets/PS3_Vid81.avi'
    results_path = '../assets/results'
    try:
        opts, args = getopt.getopt(argv, 'hv:p:', ['video=', 'path='])
    except getopt.GetoptError:
        print 'results_validator.py -v <videofile> -p <results_path>'
        sys.exit()
    for opt, arg in opts:
        if opt == '-h':
            print 'haar_sampler.py -v <videofile> -p <results_path>'
            sys.exit()
        elif opt in ('-v', '--video'):
            video_file_path = arg
        elif opt in ('-p', '--path'):
            results_path = arg

    # Initial checks.
    if not os.path.isfile(video_file_path):
        print 'Error - the specified video does not exist: ' + video_file_path
        sys.exit()
    if not os.path.isdir(results_path):
        print 'Error - the specified directory does not exist: ' + results_path
        sys.exit()
    return video_file_path, results_path

def main(argv):
    ''' Parses the command line parameters, loads the processed video to be
        analysed, keeps prompting the user if
        the results need to be accepted for each frame and stores the results
            in a CSV file.
        :param argv: command line parameters.
    '''
    # Parse the command line parameters.
    video_file_path, results_path = parse_arguments(argv)

    print '\n--------------------------------- TFM - Pere Pares Casellas
       ---------------------------------\n'
    print ' Instructions: \n'
    print '  - Move the trackbars to mark whether you accept or not each result
       .\n  - Press "n" to advance to the next frame.'

    try:
        # Image window and trackbars.
        cv2.namedWindow('frame')
        cv2.createTrackbar('Rear limb ', 'frame', 0, 1, lambda x: None)
        cv2.createTrackbar('Front limb', 'frame', 0, 1, lambda x: None)

        # Open the results file.
        results_file = open(results_path + time.strftime('/validation_%Y%m%d%H%M%S.csv', time.localtime()), 'wb')
        csv_file = csv.writer(results_file, delimiter=';')
        csv_file.writerow(['#' + video_file_path])
        csv_file.writerow(['frame', 'rear limb', 'front limb'])

        # Load the video to be analysed.
        stop = False
        cap = cv2.VideoCapture(video_file_path)
        print 'Analysing ', str(int(cap.get(CV_CAP_PROP_FRAME_COUNT))), ' frames.'
        n_frames = 0
        n_rear = 0
        n_front = 0
        while cap.isOpened():
            retrn, img = cap.read()
            if retrn == True:
                while True:
                    cv2.imshow('frame', img)
                    k = 0xFF & cv2.waitKey(0)
                    # Key bindings
                    if k == 27:            # esc to exit
                        stop = True
                        break
                    elif k == ord('n'): # Advance to the next frame.
                        n_frames += 1
                        rear = cv2.getTrackbarPos('Rear limb ', 'frame')
                        front = cv2.getTrackbarPos('Front limb', 'frame')
                        csv_file.writerow([n_frames, rear, front])
                        n_rear += rear
                        n_front += front
                        break
            else:
                break
            if stop:
                break
        print 'Results: '
        print 'Accepted rear limbs  : ' + str(n_rear*100/n_frames) + '%'
        print 'Accepted front limbs   : ' + str(n_front*100/n_frames) + '%'
    except Exception as e:
        print 'Exception thrown: ', e
    finally:
        # Release everything once the process is complete.
        cv2.destroyAllWindows()
        cap.release()
        results_file.close()

if __name__ == '__main__':
    main(sys.argv[1:])
```