CREATIVE PROGRAMMING AND COMPUTING

Lab 5:

Cognitive Agents

Luca Comanducci

luca.comanducci@polimi.it

CONTENTS

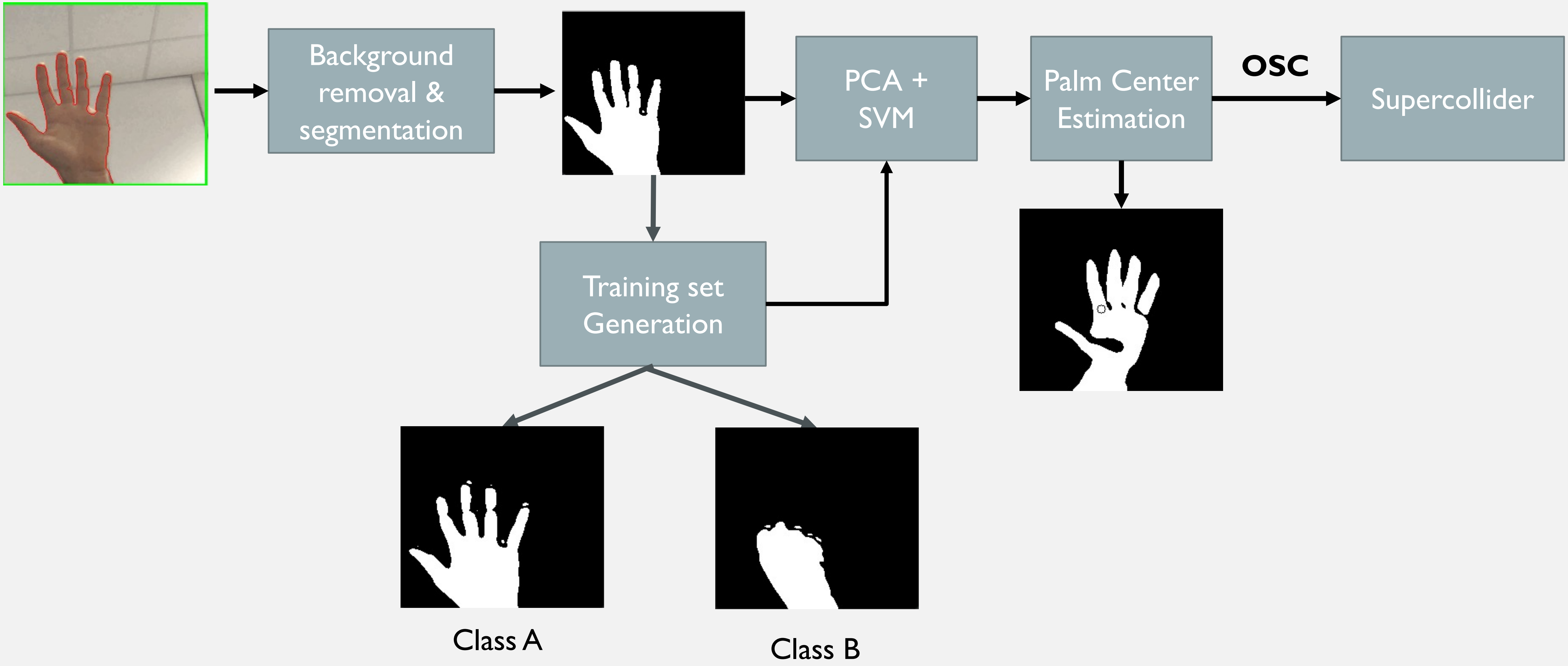
TOOLS

- **OpenCV**
  - Open source computer vision library
  - Available for several programming languages, we'll use OpenCV-Python
- **scikit-learn**
  - Data mining and data analysis library
- **python-osc**
  - Open sound control server and client implementation in pure python

HAND CONTROLLED WOBBLE BASS

- Hand Controlled Synthesizer
- Hand Gesture Classification via OSC
- OSC communication to control SuperCollider

HAND CONTROLLED WOBBLE BASS



PROJECT STRUCTURE

- *hand_detection.py*
    - Contains the main OpenCV code for gathering images, and sending OSC messages to supercollider
- *hand_detection_utils.py*
    - Auxiliary functions needed for hand detection procedure
- *SVM.py*
    - performs support vector machine classification
- *Wobble_bass.scd*
    - *Wobble bass synthesizer*
- *N.B. on MacOS you can’t run the code from VSCode, you can write the code there but then run it from the terminal.*

CODE STRUCTURE

- Main loop continuously running accepts four arguments in the opencv interface:

```
USAGE:
-Before training generate the images for the the two classes press "a" for class 1 and "b" for class 2:
-Press "a" to save class A images
-Press "b" to save class B images
-Press "t" to start SVM training (if a model has already been saved, it will be loaded)
-Press "s" to start sound generation (must be pressed after training)
-Press "q" to stop sound and "q" to stop image capture
```

IMAGE CAPTURE

- Image Capture
  - Code continuously grabs frames and converts them to grayscale:

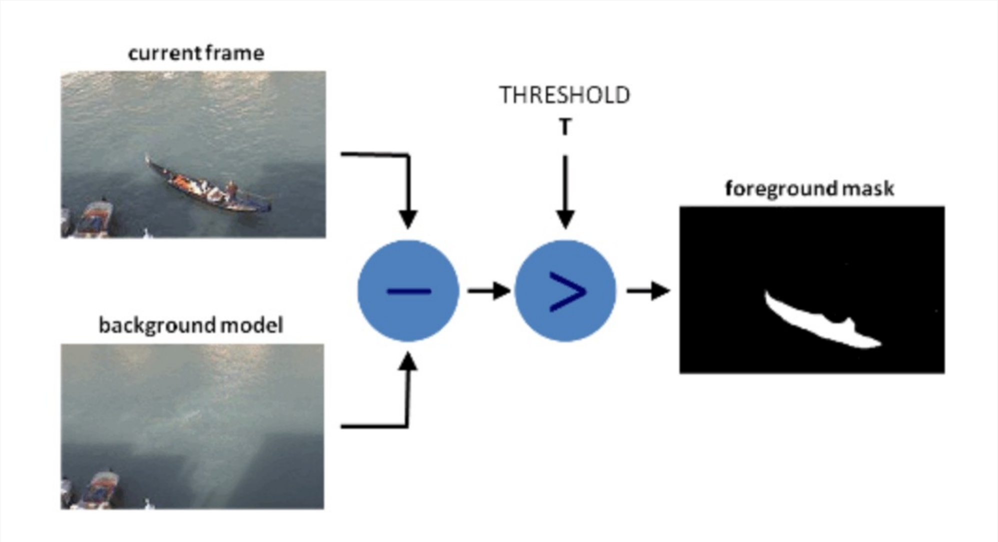
    variable gray
  - To it we are going to apply:
    - Background subtraction
    - Hand segmentation
    - Palm center detection

```
while True:
        # get the current frame
        (grabbed, frame) = camera.read()

        # resize the frame
        frame = imutils.resize(frame, width=700)

        # flip the frame so that it is not the mirror view
        frame = cv2.flip(frame, 1)

        # clone the frame
        clone = frame.copy()
        # get the height and width of the frame
        (height, width) = frame.shape[:2]
        # get the ROI
        roi = frame[top:bottom, right:left]

        # convert the roi to grayscale and blur it
        gray = cv2.cvtColor(roi, cv2.COLOR_BGR2GRAY)
        gray = cv2.GaussianBlur(gray, (7, 7), 0)
        # to get the background, keep looking till a threshold is reached
        # so that our running average model gets calibrated
        if num_frames < 30:
                run_avg(gray, aWeight)
        else:
```

BACKGROUND SUBTRACTION

- Running average over frames = background model
- Subtraction of background model from current frame
- Thresholding gets a binary representation of the moving parts

```python
# hand_detection.py
while True:
    #...
    if num_frames < 30:
        run_avg(gray, aWeight)
    else:
```




```python
# hand_detection_utils.py

def run_avg(image, a_weight):
    global bg
    # Initialize the background

    if bg is None:

        bg = image.copy().astype("float")

    # compute weighted average, accumulate it and update the background

    cv2.accumulateWeighted(image, bg, a_weight)
```

BACKGROUND SUBTRACTION

- We need to modify the segment function in order to be able to divide hand from background

```
# hand_detection.py
#...
# to get the background, keep looking till a threshold is reached
# so that our running average model gets calibrated
if num_frames < 30:
      run_avg(gray, aWeight)
else:
      # segment the hand region
      #hand = segment(gray)
      # check whether hand region is segmented
      if hand is not None:
      # if yes, unpack the thresholded image and
            # segmented region
            (thresholded, segmented) = hand

      # draw the segmented region and display the frame
      cv2.drawContours(clone, [segmented + (right, top)], -1, (0, 0,
      255))
      # Center of the hand
      #c_x, c_y = detect_palm_center(segmented)
      #radius = 5
      #cv2.circle(thresholded, (c_x, c_y), radius, 0, 1)
      cv2.imshow("Thesholded", thresholded)

      # draw square surrounding segmented hand
      cv2.rectangle(clone, (left, top), (right, bottom), (0, 255,
      0), 2)
```

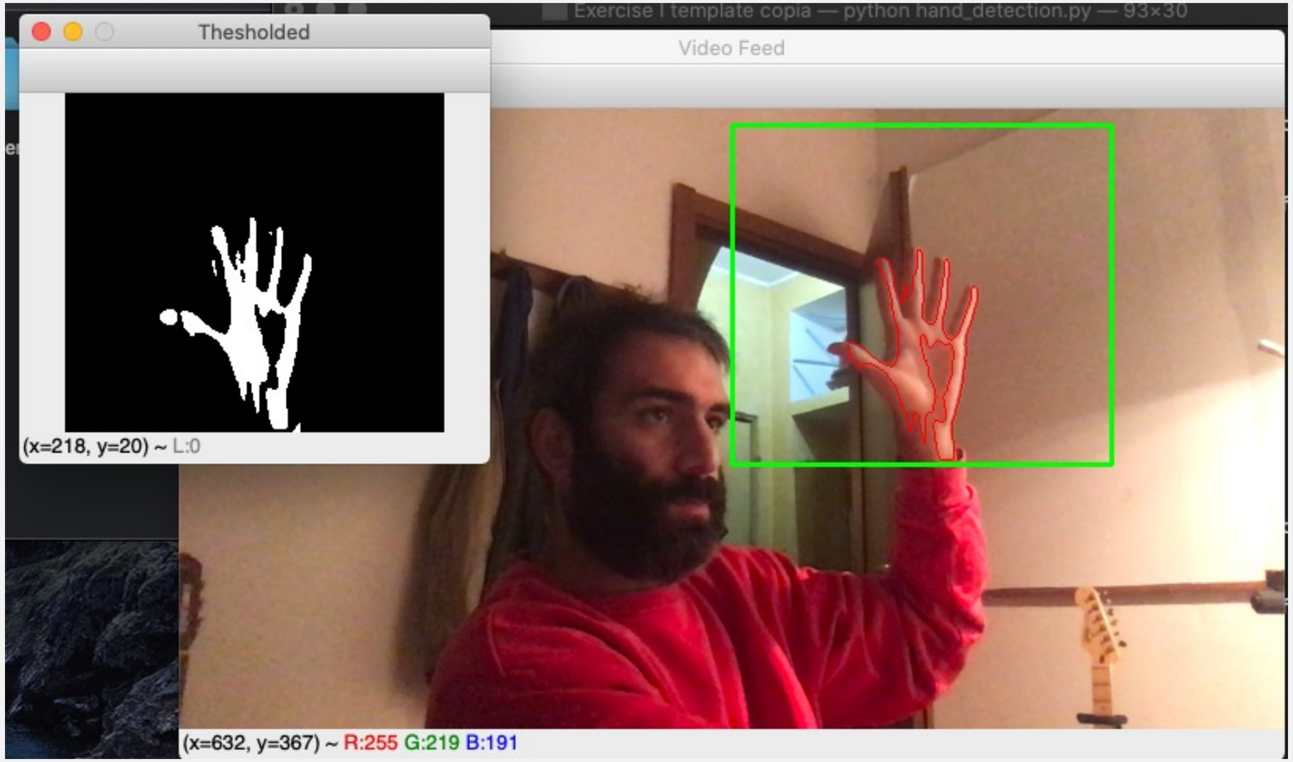
BACKGROUND SUBTRACTION

- We need to modify the segment function in order to be able to divide hand from background
- Subtract image from background
- Threshold the diff. image so that we get the foreground i.e. the hand
  - `if len(cnts) == 0:`
  - `        return`
  - `else:`
  - `        segmented = max(cnts, key=cv2.contourArea)`

```
# hand_detection_utils.py
#...
def segment(image, threshold=25):
    global bg

    # Find the absolute difference between background and current
    image
    # FILL THE CODE
    # diff = cv2.absdiff(bg.astype("uint8"), # ....? )
    # Threshold the diff image so that we get the foreground
    # FILL THE CODE
    #thresholded = cv2.threshold(# ....?, # ....?, 255,
    cv2.THRESH_BINARY)[1]

    # get the contours in the thresholded image

    (cnts, _) = cv2.findContours(thresholded.copy(),
    cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # return None, if no contours detected (contours is a list)
    # otherwise return:
    # - the segmented contour as the maximum one w.r.t. to the area
    # - the thresholded image
    # HINT: to the max() function you can pass an argument as key
    #where the iterables are passed and comparison is performed
    based on its return value (use cv2.contourArea as key)

    if # FILL the code:
    #     FILL the code
    # else
    #     FILL the code

    return thresholded, segmented
```

BACKGROUND SUBTRACTION

- This is what you should get now



PALM CENTER EXTRACTION

- Now we want to extract the palm center of the segmented hand in order to use it to control the synthesizer

- Uncomment and complete these lines

  and open *hand_detection_utils.py*

```
# hand_detection.py
#...
# to get the background, keep looking till a threshold is reached
# so that our running average model gets calibrated
if num_frames < 30:
      run_avg(gray, aWeight)
else:
      # segment the hand region
      hand = segment(gray)
      # check whether hand region is segmented
      if hand is not None:
      # if yes, unpack the thresholded image and
            # segmented region
            (thresholded, segmented) = hand

      # draw the segmented region and display the frame
      cv2.drawContours(clone, [segmented + (right, top)], -1, (0, 0, 255))
      # Center of the hand
      #c_x, c_y = detect_palm_center(segmented)
      #radius = 5
      #cv2.circle(# image where we draw the circle, # tuple representing center,radius, 0, 1)
      cv2.imshow("Thesholded", thresholded)

      # draw square surrounding segmented hand
      cv2.rectangle(clone, (left, top), (right, bottom), (0, 255, 0), 2)
```

PALM CENTER EXTRACTION

- *detect_palm_center(segmented)*

  computes the (approximate) center of the hand

- The extreme points of the hand are computed through the convex hull

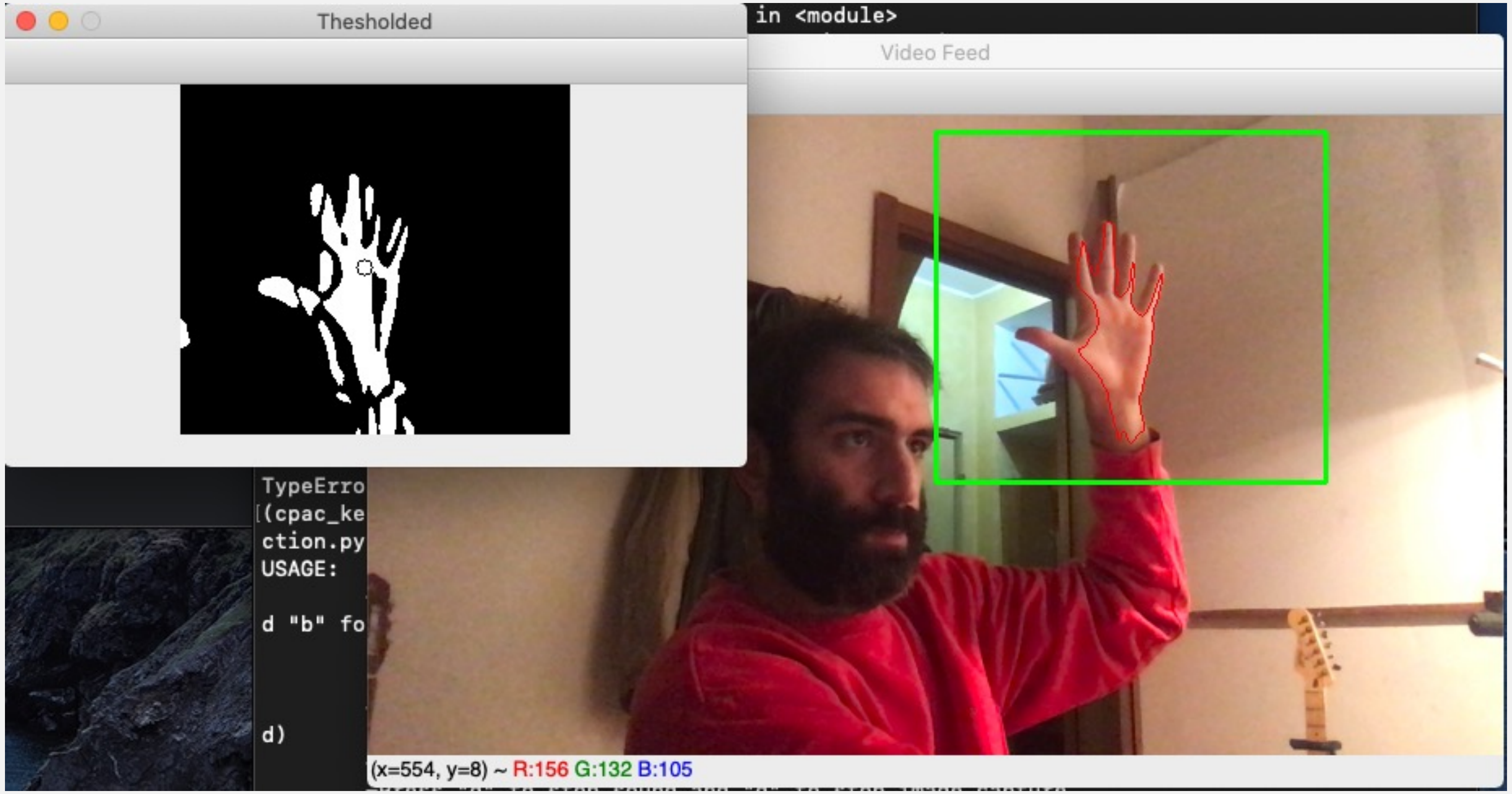
- Using these extreme points compute the center of the hand coordinates

  N.B. the function must return integer coordinates (we are working with pixels)

```
# hand_detection_utils.py

def detect_palm_center(segmented):
      # Find the convex hull of the segmented hand region
      chull = cv2.convexHull(segmented)

      # Find the most extreme points in the convex hull
      extreme_top = tuple(chull[chull[:, :, 1].argmin()][0])[1]
      extreme_bottom = tuple(chull[chull[:, :, 1].argmax()][0])[1]
      extreme_left = tuple(chull[chull[:, :, 0].argmin()][0])[0]
      extreme_right = tuple(chull[chull[:, :, 0].argmax()][0])[0]


      # Find the center of the palm
      #c_x = # FILL THE CODE
      #c_y = # FILL THE CODE
return c_x, c_y
```

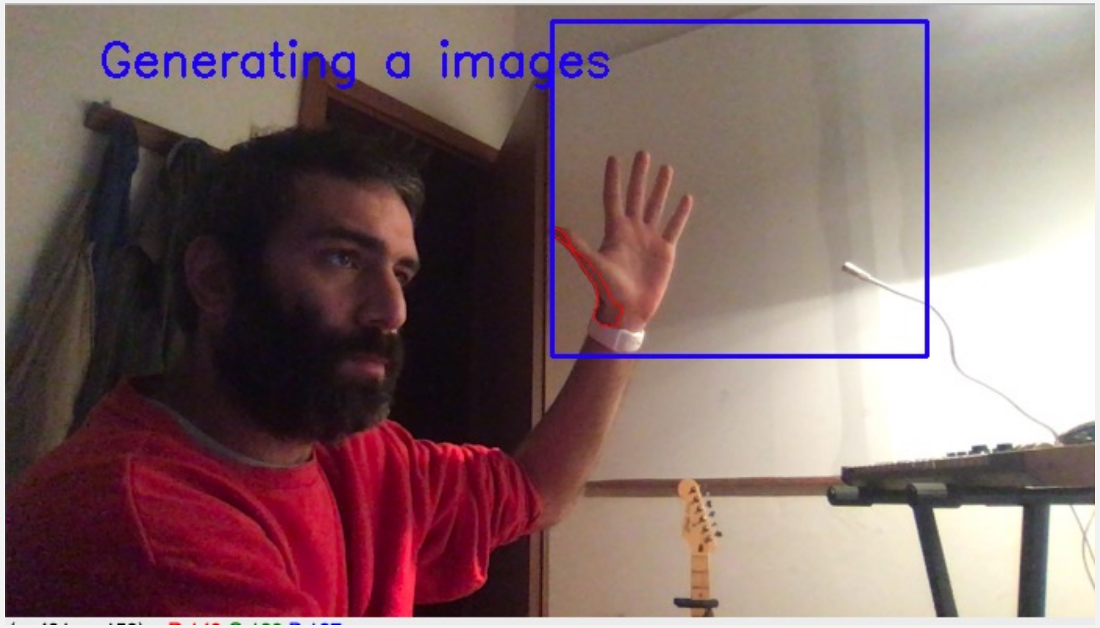
BACKGROUND SUBTRACTION

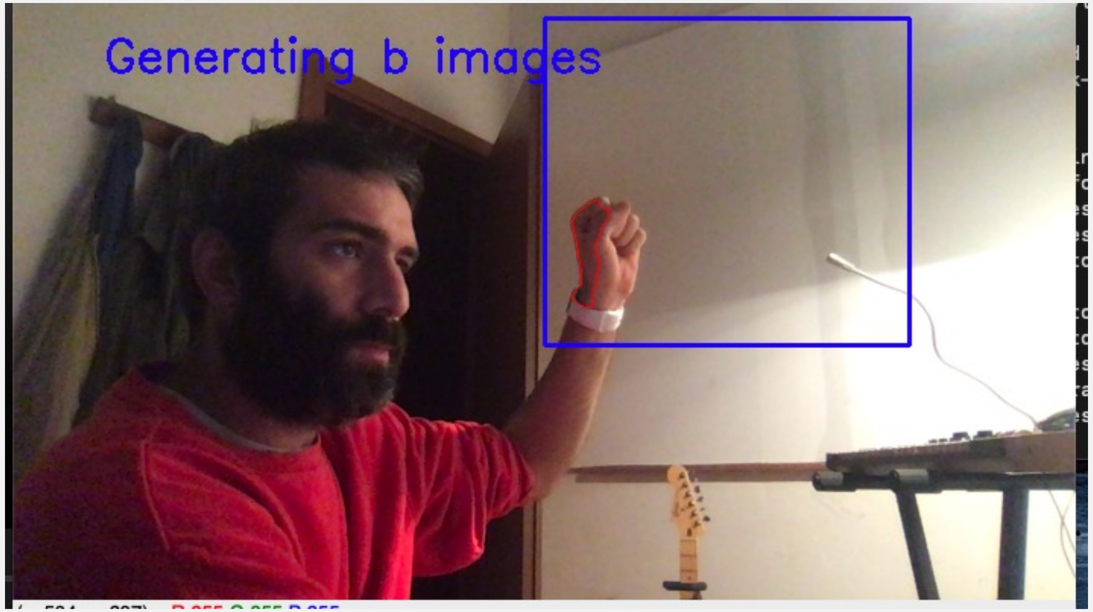
- This is what you should get now



GENERATE TRAIN SET

- *In hand_detections.py*
  - When we press ‘a’ or ‘b’ the corresponding image is saved





GENERATE TRAIN SET

- *In hand_detections.py*
  - When we press ‘a’ or ‘b’ the corresponding image is saved
  - We keep track of how many images are captured using

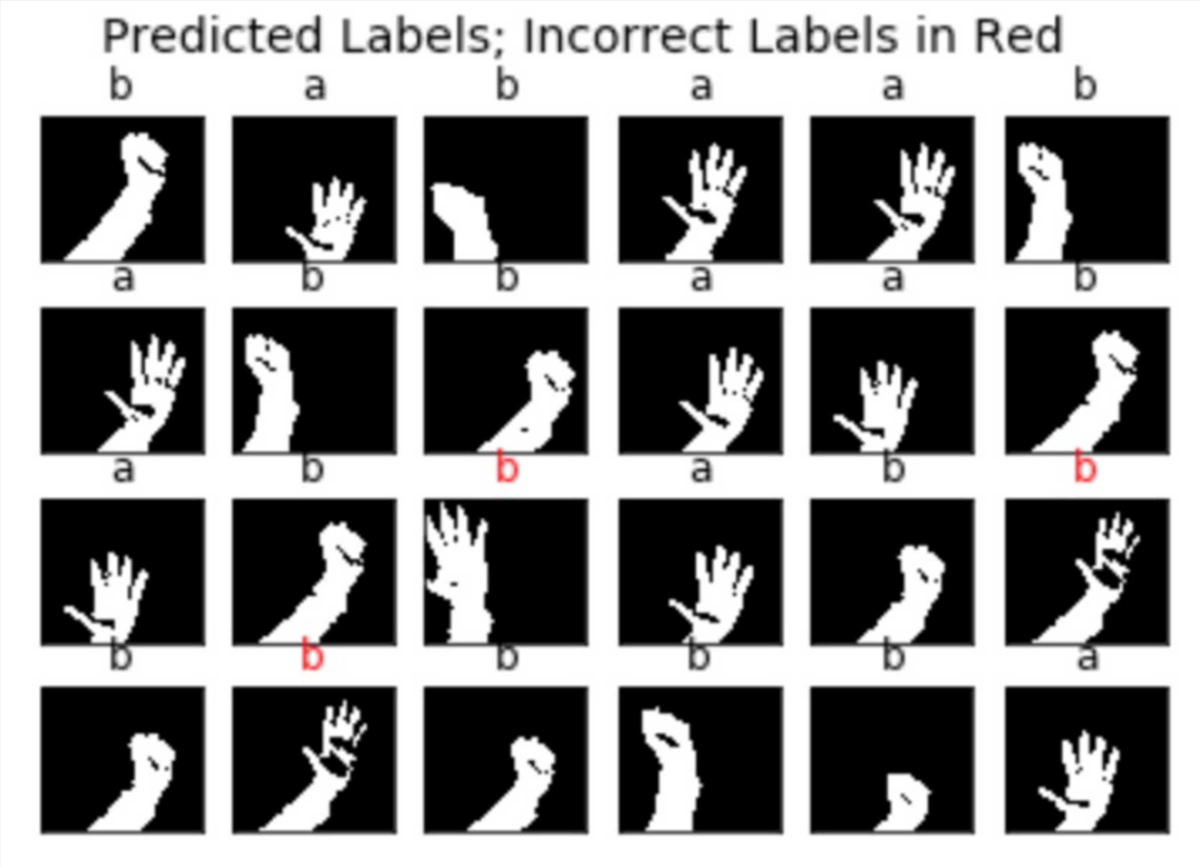
  num_frames

```
# Generate class A images
if keypress == ord("a"):
        print('Generating the images for class A:')
        TRAIN = True
        num_frames_train = 0
        tot_frames = 250
        class_name = 'a'

# Generate class B images
if keypress == ord("b"):
        print('Generating the images for class B:')
        TRAIN = True
        num_frames_train = 0
        tot_frames = 250
        class_name = 'b'
```

```
# increment the number of frames
num_frames += 1
# We want to generate and save the images corresponding to the two classes, in order to then save the model

if TRAIN:

        #Check if directory for current class exists
        if not os.path.isdir('images/class_'+class_name):
                os.makedirs('images/class_'+class_name)

        if num_frames_train < tot_frames:
                # Change rectangle color to show that we are saving training images
                cv2.rectangle(clone, (left, top), (right, bottom), (255, 0, 0), 2)
                text = 'Generating ' + str(class_name) + ' images'
                cv2.putText(clone, text, (60, 45), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 0, 0),
                2)

                # Save training images corresponding to the class
                cv2.imwrite('images/class_'+class_name+'/img_'+str(num_frames_train)+'.png',
                thresholded)

                # keep track of how many images we are saving
                num_frames_train += 1

        else:

                print('Class '+class_name+' images generated')
                TRAIN = False
```

SVM

- Open *SVM.py*
  - This code:
    - Loads the dataset
    - Trains a SVM model
    - Performs classification



SVM

- Open *SVM.py*
  - It contains two functions:
    - load_data(class_a_path, class_b_path)
      - No need to worry about it, simply reads the images and converts them into numpy arrays
      - Returns:
        - Images→ 2D arrays containing images
        - Images_vector → unrolled image in 1D array
        - Labels→ class labels
    - train_svm()
      - Computes the SVM model

N.B. the first time that you train the model it will take some time and the video interface could be blocked for several minutes! Later you can simply reload the training model that is automatically saved.

EXERCISE I - SVM

- Open *SVM.py*
- Fill in the missing parts

  - Combine PCA and SVM
  - Hint: use `make_pipeline(*steps list of estimators.)`

  - Split into training and test set

  hint: use [train_test_split]

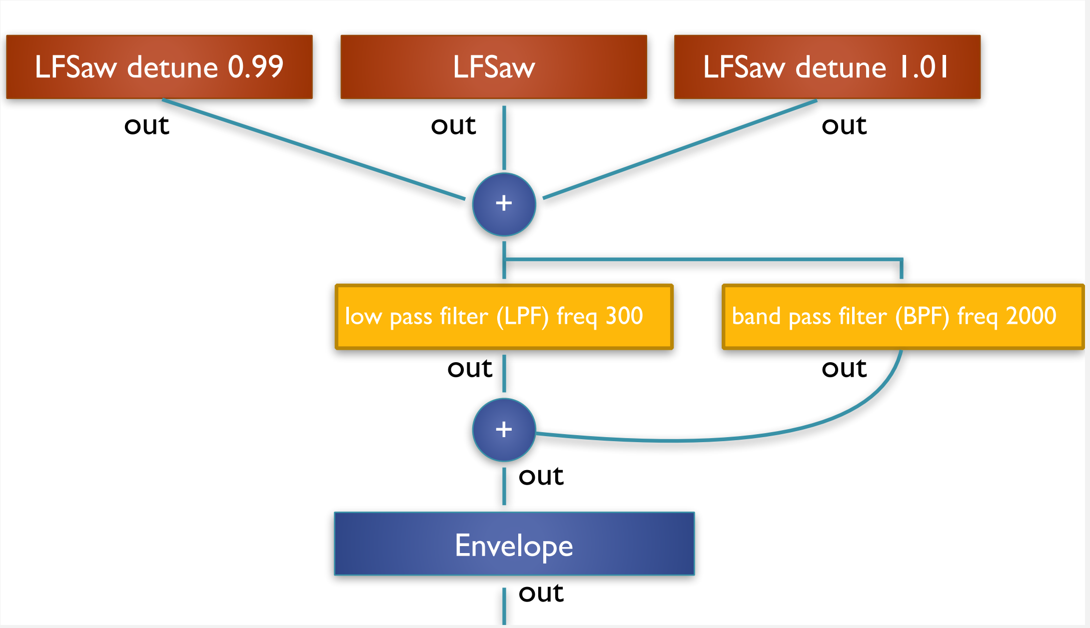
  (n.b. here is useless, but useful if you want to test it)

  - Fit the SVM model using params defined in grid
    - Hint: `grid.fit(x,y)`
- Once the code is ready you can press ''t'' in the opencv window to start training (It will take some time)
- https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

```
def train_svm():
    # Load data
    class_names = ['a', 'b']

    images, images_vector, labels = load_data(class_a_path='images/class_a/',
    class_b_path='images/class_b/')

    pca = PCA(n_components=150, svd_solver='randomized', whiten=True,
    random_state=42)
    svc = SVC(kernel='rbf', class_weight='balanced')
    #model = # FILL THE CODE

    with warnings.catch_warnings():
        # ignore all caught warnings
        warnings.filterwarnings("ignore")
        # Split in training and test set
        xtrain, xtest, ytrain, ytest = # FILL THE CODE

    param_grid = {'svc__C': [1, 5, 10, 50], 'svc__gamma': [0.0001, 0.0005, 0.001,
    0.005]}
    grid = GridSearchCV(model, param_grid)

    print('Fit the SVM model')
    #FILL THE CODE

    print(grid.best_params_)

    model = grid.best_estimator_

    # Save the model
    dump(model, 'modelSVM.joblib')

return model
```

SYNTHESIZER CONTROL

- When the SVM training has finished:
  - We can finally start generating sound (press ''s'' in the opencv interface)
  - Center of the palm provides us two coordinates
  - We use these to map values to supercollider via OSC
  - Open *synth.scd* , which contains the Wobble bass synth from *COMPUTER MUSIC – LANGUAGES AND SYSTEMS* course



SYNTHESIZER CONTROL

OSCFunc manages OSC communication between SC and python

Complete the code in hand_detection.py

When **CLASS == 0**
- X coordinate -> main frequency
- Y coordinate -> amplitude

When **CLASS == 1**
- X coordinate-> lfo
- Y coordinate-> detuning

```
# Here we send the OSC message corresponding
#if START_SOUND:
        #if class_test == 0:

        #freq = # FILL THE CODE
        #amp = # FILL THE CODE
        #client.send_message(# FILL THE CODE)
#else:
        #detune = # FILL THE CODE
        #lfo = # FILL THE CODE
        #client.send_message(# FILL THE CODE)
```

- Use the palm center coordinates and the width/height of the ROI to control the synth parameters
  - freq → frequency between 0 and 100 (not really frequency but MIDI note)
  - amp → amplitude, leave it between 0 and 1
  - detune → detuning of synth freqs, between 0 and 0.1
  - lfo → low frequency oscillator, between 0 and 10

- Message should contain: first argument: synth name, second argument list with [class, param1, param2]

SYNTHESIZER CONTROL

- OSCFunc manages OSC communication between SC and python
    - When **CLASS == A**
        - X -> main frequency
        - Y -> amplitude
    - When **CLASS == B**
        - X -> lfo
        - Y -> detuning

- Complete the function
  - N.B. values must be converted correctly in order to be used by SC

    hint `.asFloat()`

```
(
x = OSCFunc( { | msg, time, addr, port |
    var pyFreq,pyAmp,pyDetune,pyLfo;

    // Handle end of sound
    if (msg[1] =='stop'){
        h.free
    }
    {
        // handle class A message (freq and amplitude)
        if (msg[1]=='a'){
            // Parse message
            pyFreq = //...
            pyAmp  = //...
            ( "freq is " + pyFreq ).postln;
            ( "amp is " + pyAmp ).postln;

            // set parameters
             h.set( \note, pyFreq );
             h.set( \amp, pyAmp );
        };
        // handle class B message (detune and lfo)
        if (msg[1]=='b'){
            // parse message
            pyDetune = //...
            pyLfo = //...

            // print info
            ( "Detuning is " + pyDetune ).postln;
            ( "lfo is " + pyLfo ).postln;

            // set parameters
             h.set( \detune, pyDetune );
             h.set( \lfo, pyLfo );
        };

    };

}, '/synth_control' );
)
```

MARKOV CHAIN FOR MUSIC GENERATION

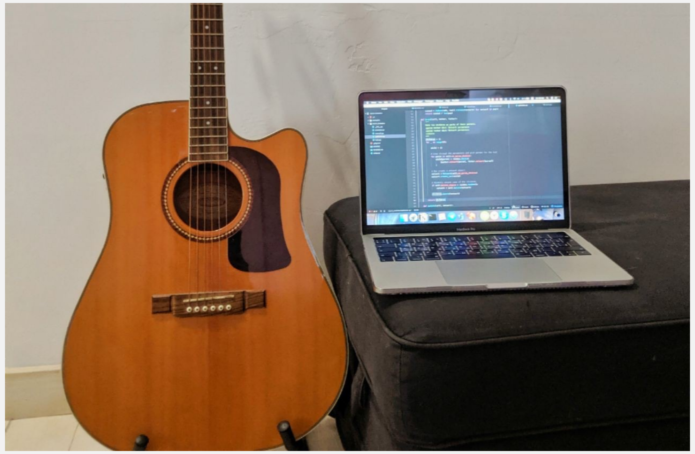
- Markov chain for music Generation
- Model markov chain using a corpus of beatles chords
- Play the chords on supercollider

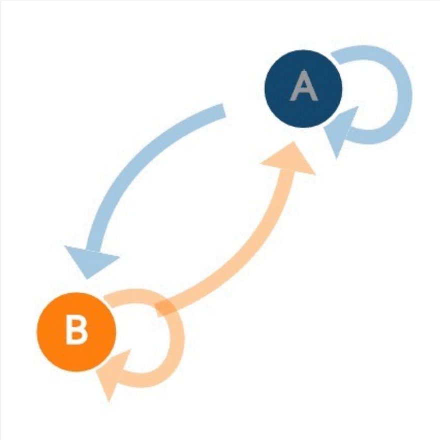
MARKOV CHAIN FOR MUSIC GENERATION

MARKOV CHAIN FOR MUSIC GENERATION

- Idea:
  - Take a chorpus of chords of beatles' songs

  - Generate new chord sequences through markov chains





MARKOV CHAIN FOR MUSIC GENERATION

- How do we do it?
    1. Take chorpus of chords
    2. Calculate probability distribution for chords to follow a particular chord
    3. Define chord where to start (this could be done arbitrarily)
    4. Make random choice for next chord taking into account the probability distribution
    5. Repeat steps 1-4 for each new chord that you want to insert in the sequence

MARKOV CHAIN FOR MUSIC GENERATION

- How do we do it?
  1. Take chorpus of chords
  2. Calculate probability distribution for chords to follow a particular chord
  3. Define chord where to start (this could be done arbitrarily)
  4. Make random choice for next chord taking into account the probability distribution
  5. Repeat steps 1-4 for each new chord that you want to insert in the sequence

MARKOV CHAIN FOR MUSIC GENERATION

- Let's Do it step by step:
  - Consider a chords sequence

    ['F', 'Em7', 'A7', 'Dm', 'Dm7', 'Bb', 'C7', 'F', 'C', 'Dm7',...]

  - We can make bigrams out of it

```
# Main.py
# Generate Bigrams
n = 2
chords = data['chords'].values
ngrams = zip(*[chords[i:] for i in range(n)])
bigrams = [" ".join(ngram) for ngram in ngrams]

bigrams[:5]
```

Run it using Python Interactive

['F Em7', 'Em7 A7', 'A7 Dm', 'Dm Dm7', 'Dm7 Bb', 'Bb C7', ...]

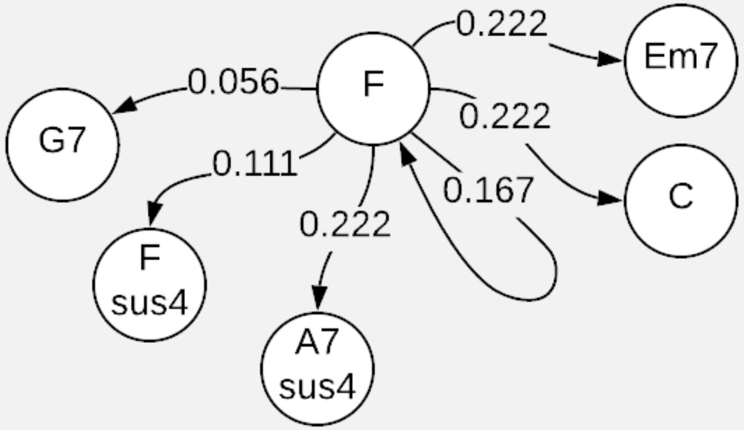
MARKOV CHAIN FOR MUSIC GENERATION

- Now we want to modify the *predict_next_state(…)* function in order to compute probability of a chord followed by another apply markov chains:
  - If we compute the frequency of each unique bigram appearing in the sequence we get

    'F Em7': 4, 'F C': 4, 'F F': 3, 'F A7sus4': 4, 'F Fsus4': 2, 'F G7': 1

  - If we normalize it, we can get the probability of each bigram, which can be interpreted as a markov chain graph

'F Em7': 0.222,
'F C': 0.222,
'F F': 0.167,
'F A7sus4': 0.222,
'F Fsus4': 0.111,
'F G7': 0.056



MARKOV CHAIN FOR MUSIC GENERATION

- Try to complete the function considering *chord='F'* and taking advantage of the Python Interactive Interface on VSCode

```
# Main.py
def predict_next_state(chord:str, data:list=bigrams):
    """Predict next chord based on current state."""
    # create list of bigrams which stats with current chord
    bigrams_with_current_chord = #FILL CODE

    # count appearance of each bigram
    count_appearance = #FILL CODE

    # convert apperance into probabilities
    for ngram in count_appearance.keys():
        count_appearance[ngram] = # FILL CODE
    # create list of possible options  for the next chord
    options = #FILL CODE
    # create list of probability distribution
    probabilities = #FILL CODE
    # return random prediction
    return #FILL CODE
```

- Computes bigrams starting with selected chord

['F Em7', 'F C', 'F F', 'F Em7',….

- *count_appearance* must contain dictionary with how many times a certain bigram appears

e.g.
{'F Em7': 4, 'F C': 4, 'F F': 3, 'F A7 sus4': 4, 'F Fsus4': 2, 'F G7': 1}

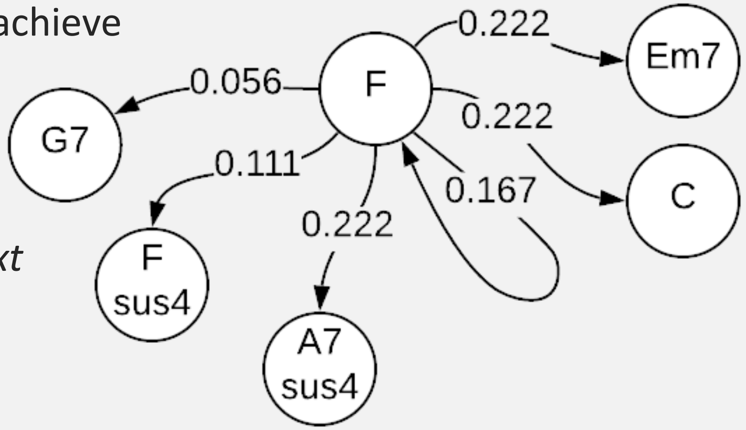
- The value of count_appearance should be changed in order to contain probability instead of number of appearance:

{'F Em7': 0.2222222222222222,
'F C': 0.2222222222222222,
'F F': 0.16666666666666666,
'F A7sus4': 0.2222222222222222,
'F Fsus4': 0.1111111111111111,
'F G7': 0.05555555555555555}

- Return next chord selecting it from option using the computed probabilities, hint: np.random.choice

MARKOV CHAIN FOR MUSIC GENERATION

- Each node of this graph represents possible states that our sequence can achieve
- In this specific case: chords following *F*



- *N.B. Markov chain is a stochastic process, we will choose randomly the next chord by following the derived probability distribution from the graph*

- *By repeating this process iteratively, we can generate sequences as long as we want e.g.*

'Bb','Dm','C','Bb','C7','F','Em7','A7','Dm','Dm7','Bb','Dm','Gm6'

MARKOV CHAIN FOR MUSIC GENERATION

- Now complete the *generate_sequence* function that starting from one chord generates a sequence

```
# Main.py
def generate_sequence(chord:str=None, data:list=bigrams, length:int=30):
"""Generate sequence of defined length."""
# create list to store future chords
chords = []
for n in range(length):
      # append next chord for the list
      # FILL CODE
      # use last chord in sequence to predict next chord
      # FILL CODE
return chords
```

Predict next chord taking advantage of the *predict_next_state* function

MARKOV CHAIN FOR MUSIC GENERATION

- Now we can simply generate a sequence of chords using the command

```
chords = generate_sequence('C')
```

Pay attention to port number, check it in SC using *NetAddr.langPort*

- We can play the chords starting the communication with supercollider

```
def start_osc_communication():
      # argparse helps writing user-friendly commandline interfaces
      parser = argparse.ArgumentParser()
      # OSC server ip
      parser.add_argument("--ip", default='127.0.0.1', help="The ip of the OSC server")
      # OSC server port (check on SuperCollider)
      parser.add_argument("--port", type=int, default=57121, help="The port the OSC server is listening on")

      # Parse the arguments
      args = parser.parse_args()

      # Start the UDP Client
      client = udp_client.SimpleUDPClient(args.ip, args.port)

      return client

client = start_osc_communication()
```

MARKOV CHAIN FOR MUSIC GENERATION

- Send generated chords via OSC to supercollider
  - We distinguish 3 vs 4 note chords

```
# Send chords
for c in chords:
print(c)
if len(chords_midi_dict[c]) == 3:
      client.send_message("/synth_control",['chord3',chords_midi_dict[c][0],chords_midi_dict[c][1],chords_midi_dict[c][2]])
time.sleep(1)
if len(c) == 4:
      client.send_message("/synth_control",['chord4',chords_midi_dict[c][0],chords_midi_dict[c][1],chords_midi_dict[c][2],chords_midi_dict[c][3]])
time.sleep(1)
```

Chords are converted in MIDI before being sent to supercollider

```
chords_midi_dict={
'F':[5,9,12],
'Em7':[4,9,11,14],
'A7':[9,13,16,21],
'Dm':[2,5,9],
'Dm7':[2,5,9,12],
'Bb':[10,14,17],
'C7':[0,4,7],
'C':[0,4,7,10],
'G7':[7,11,14,17],
'A7sus4':[9,14,16,21],
'Gm6':[7,10,14,16],
'Fsus4':[5,10,12],
}
```

MARKOV CHAIN FOR MUSIC GENERATION

- Supercollider receives OSC message and plays corresponding chords
- Pbinds handles the chord generation

```
(
x = OSCFunc( { | msg, time, addr, port |
    var
chord,note1,note2,note3,note4,pyFreq,pyAmp,pyDetune,pyLfo;

      // Handle end of sound
      if (msg[1] =='stop'){
            h.free
      }
      {
      // handle class A message (freq and amplitude)
            if (msg[1]=='chord3'){
                  // Parse message
                  note1 = msg[2].asFloat;
                  note2  = msg[3].asFloat;
                  note3  = msg[4].asFloat;
                  chord=[note1,note2, note3];
                  chord.postln();
            };
            if (msg[1]=='chord4'){
                  // Parse message
                  note1 = msg[2].asFloat;
                  note2  = msg[3].asFloat;
                  note3  = msg[4].asFloat;
                  note4  = msg[5].asFloat;
                  chord=[note1,note2, note3,note4];
                  chord.postln();
            };
            (
                  p=Pbind(
                        \instrument, \harpsi,
                        \note, Pseq([chord],1),
                        \dur, 1,
                        \legato, 0.4,
                        //\strum, 0.1 // try 0, 0.1, 0.2, etc
                  ).play;
            )
      };

}, '/synth_control' );
)
```

REFERENCES

- Background extraction and hand segmentation:
  - https://gogul.dev/software/hand-gesture-recognition-p1
  - https://gogul.dev/software/hand-gesture-recognition-p2
- Support Vector Machines:
  - https://towardsdatascience.com/support-vector-machine-python-example-d67d9b63f1c8
- Music generation using Markov Chains
  - https://towardsdatascience.com/markov-chain-for-music-generation-932ea8a88305