Computer Vision: Open CV

Manual Template

Author: Layth Ayache

Contents

[1. Introduction 3](#_Toc200282161)

[2. Image Features and Alignment 3](#_Toc200282162)

[2.1 theory of alignment 3](#_Toc200282163)

[2.2 How to estimate Homography 4](#_Toc200282164)

[3. Getting Started with Images 6](#_Toc200282165)

[3.1 Loading & Display 6](#_Toc200282166)

[3.2 Image Properties 8](#_Toc200282167)

[4. Basic Image Manipulation 11](#_Toc200282168)

[4.1 Color Channels 11](#_Toc200282169)

[4.2 Color Conversions 12](#_Toc200282170)

[5. Image manipulation 13](#_Toc200282171)

[5.1 Accessing Individual Pixel 13](#_Toc200282172)

[5.2 Modifying Image Pixels 13](#_Toc200282173)

[5.3 Cropping images 14](#_Toc200282174)

[5.3 Resizing Images 14](#_Toc200282175)

[5.4 Flipping Images 15](#_Toc200282176)

[6. Image Annotation 16](#_Toc200282177)

[6.1 Drawing a Line 16](#_Toc200282178)

[6.2 Drawing a Circle 16](#_Toc200282179)

[6.3 Drawing a Rectangle 17](#_Toc200282180)

[6.4 Adding Text 17](#_Toc200282181)

[7. Image Enhancement 19](#_Toc200282182)

[7.1 Brightness 19](#_Toc200282183)

[1.2 Contrast 19](#_Toc200282184)

[1.3 Image Thresholding 20](#_Toc200282185)

[1.4 bitwise Operations 23](#_Toc200282186)

[2. Accessing the Camera 29](#_Toc200282187)

[3. Video Writing 30](#_Toc200282188)

[9.1 Read Video from Source 30](#_Toc200282189)

[3.2 Read and display one frame 30](#_Toc200282190)

[3.3 Write Video 31](#_Toc200282191)

[4. Image Filtering (Edge Detection) 32](#_Toc200282192)

[Conclusion 37](#_Toc200282193)

# 1. Introduction

The objective of this report is to showcase the full extent of the OpenCV library, and to guide any person that reads this report on how to use it and do the following :

1. Image Features and Alignment
2. Panorama
3. HDR
4. Object Tracking
5. Face Detection
6. TensorFlow Object Detection
7. Pose Estimation Using OpenPose

The content of this report were directly taken from the OpenCV bootcamp course :” <https://courses.opencv.org/courses/course-v1:OpenCV+Bootcamp+CV0/course/> “

# 2. Image Features and Alignment

## 2.1 theory of alignment



A group of squares with black text

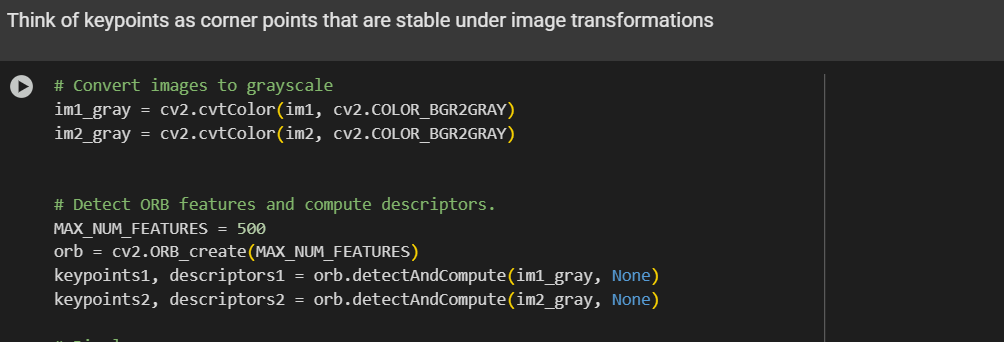
AI-generated content may be incorrect.

Think of it like this: **translation** is just swiping the square up/down/left/right; **Euclidean** lets you swipe and spin it without changing its size; **similarity** adds uniform zooming (so you can rotate, swipe, and push it closer or farther, but it stays proportional); **affine** cranks it up a notch—you can rotate, swipe, stretch or squash it differently in x and y, and even skew it like a parallelogram; and **homography** is the full 3D‐tilt cheat—everything above plus real perspective, so you can lay it down or angle it back and make parallel lines meet.

## 2.2 How to estimate Homography

Images of two planes are related by a homography, meaning that two images of the same object but with different perspectives are relate, and we can estimate Homography using 4 corresponding points.  
To estimate a homography between two images of the same planar surface, you:

1. **Detect & describe keypoints** in each image (e.g. using ORB, SIFT or AKAZE) to get two sets of points and descriptors.  
   PS: Orb and Surf and SIFT are feature extractors, they try to find similar point in the images, they are a built in function of OpenCV and are very efficient at comparing two images or two frames and describe keypoints (hence the name descriptors)
2. **Match descriptors** across images (e.g. with a brute-force Hamming/FLANN matcher) and filter out poor matches (e.g. keep the top 10 % by distance or use Lowe’s ratio test).
3. **Assemble point correspondences** from the inlier matches, collecting two arrays of 2D coordinates.
4. **Run RANSAC** with OpenCV’s cv2.findHomography(points\_src, points\_dst, cv2.RANSAC), which repeatedly:
   * Picks four random matches,
   * Computes a candidate HHH via the Direct Linear Transform (DLT),
   * Counts how many other matches agree (inliers) under a reprojection threshold,
   * And retains the best HHH.
5. **(Optional) Refine** the final estimate by minimizing reprojection error over all inliers (e.g. with nonlinear least squares).

**In code:****A screen shot of a computer program

AI-generated content may be incorrect.**A screen shot of a computer program

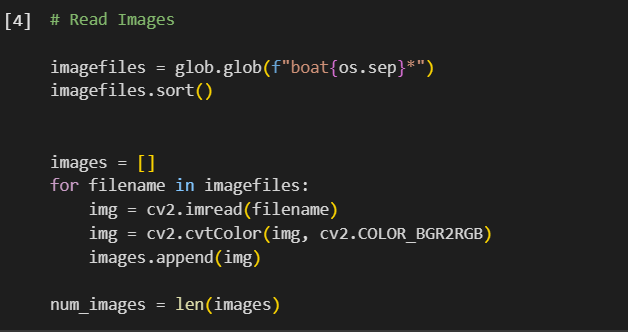
AI-generated content may be incorrect.

# 3. Panorama

## 3.1 Steps for creating Panoramas

There are 5 main steps to creating Panoramas,

* Find keypoints in all images
* Find pairwise correspondences
* Estimate pairwise Homographies
* Refine Homographies
* Stitch with Blending

IN CODE:  


A screen shot of a computer program

AI-generated content may be incorrect.

## 3.2 Cons and how to get around it



An important part to mention is the black parts of the pictures that was due to the warping made during the stitching. We can mitigate this by : Auto‐crop to the largest valid region, Use a different projection before stitching, Fill the border when warping.

# 4. High Dynamic Range Imaging (HDR)

## 4.1 Theory

High Dynamic Range (HDR) imaging extends the limited 8-bit dynamic range of standard photographs by merging multiple images taken at different exposure levels. This allows preservation of detail in both very dark and very bright regions of a scene.  
Now to understand HDR and why we do it we must first discuss the limitations we face for image processing or in photography in general:

**Dynamic Range Limitations:**

8‑bit per channel: Pixel values in each channel are clamped to [0,255].

Clipping: Bright regions saturate at 255; shadows clip to 0, losing detail.

**Exposure Bracketing:**

Capture a sequence of exposures (img\_0.033.jpg, img\_0.25.jpg, img\_2.5.jpg, img\_15.jpg).

Each image samples a different portion of the scene's luminance curve.

Exposure times stored in times = [1/30, 0.25, 2.5, 15] allow recovery of real-world intensities.

## 4.2 Image alignment (MTB)

Before we apply HDR, we must make sure that all pictures are aligned, and in real life scenarios, some movement or jitter are bound to happen, so what we do is align all the pictures using MTB: a built in opencv function, what it does is make sure that all pictures are aligned properly to the sub pixels.

Median Threshold Bitmap (MTB): Aligns images by comparing binary thresholds, robust to exposure differences.

Implemented via cv2.createAlignMTB() and alignMTB.process(images, images) to ensure pixel correspondence.

## 4.3 Camera Response Function

The **Camera Response Function (CRF)** describes how a real camera maps scene radiance (the true light intensity hitting the sensor) to recorded pixel values. In practice, sensors and in-camera processing introduce a nonlinear curve g so that a measured pixel value Z relates to scene irradiance E and exposure time Δt by:  


Recovering the **inverse CRF** g(−1) lets us linearize each image—i.e., convert its pixel values back into proportional estimates of true scene brightness—which is essential when merging multiple exposures into a single HDR radiance map. In OpenCV, Debevec’s method (createCalibrateDebevec) solves for g and the underlying radiances jointly via least squares over all pixels and exposures.

## 4.4 HDR Mapping

Imagine you have several photos of the same scene taken at different “brightness settings” (one dark, one normal, one very bright). Each photo captures detail in a different part of the scene—shadows in the bright shot, highlights in the dark shot, and mid-tones in the normal shot. **HDR merging** is simply the process of blending those photos into one picture that keeps the best parts of each:

1. **Line up** the shots so everything overlaps perfectly.
2. **Convert** each pixel’s value back into a true measure of light (undoing the camera’s built-in “curve” that squishes brightness).
3. **Combine** the pixels by giving more weight to the photo where that spot is best exposed (e.g., use the dark image for bright patches so they aren’t blown out).
4. **Result**: a single image that shows both deep shadows and bright highlights in full detail—just like how your eyes naturally adjust to see everything, from the darkest corners to the sunlit sky.

## 4.5 Tone Mapping

Think of your HDR image as having way more “brightness range” than your screen or print can show—all those extra lights and darks need to be squished back into the displayable 0–255 range without losing detail. **Tone mapping** does exactly that in four simple steps:

1. **Measure the range**  
   – See how bright your brightest pixels are and how dark the darkest are—often far outside what a normal monitor can show.
2. **Compress globally**  
   – Apply a gentle curve (like turning down the overall “volume” of brightness) so the extremes move closer together.
3. **Preserve local contrast**  
   – Within small neighborhoods, boost differences so textures and edges still pop (so the brickwork in shadows and the ripples in highlights remain visible).
4. **Clamp to display**  
   – Finally, map everything into the 0–255 bucket for each channel, yielding a “regular” LDR image that still looks rich and detailed.

In essence, tone mapping is the art of squeezing an ultra-wide range of light values into what our screens or prints can handle—keeping both the deep shadows and the dazzling highlights looking natural.

# Object Tracking

Object tracking follows an initial detection of a region of interest (ROI) and aims to continuously update that ROI across subsequent video frames. This pipeline leverages both motion and appearance models to handle translation, scale changes, occlusions, and deformations.

## Types of models

There exist two relevant types of models for tracking:

Motion Model: Predicts the new position of the ROI based on previous motion (e.g., constant velocity).

Appearance Model: Captures how the object looks (e.g., histogram, keypoints, template) so the tracker can re-detect it under changes.  
PS: you activate the appearance model by selecting manually or by using a classifier model , but the first frame should be selected so that the motion and appearance model will have a reference for the histogram to be able to keep track of the object

|  |  |  |  |
| --- | --- | --- | --- |
| **Tracker** | **Core Idea** | **Strength** | **Weakness** |
| BOOSTING | AdaBoost on image patches | Simple | Slow, limited precision |
| MIL | Multiple Instance Learning | Robust to partial occlusion | Moderate speed |
| KCF | Kernelized Correlation Filters | Good speed + accuracy | Struggles with fast occlusions |
| CSRT | Discriminative Correlation with Scale | High accuracy, scale estimation | Slower than KCF |
| TLD | Tracking-Learning-Detection | Recovers from long-term occlusions | Complex, slower |
| MEDIANFLOW | Forward-backward error consistency | Stable for slow predictable motion | Fails on fast motion |
| GOTURN | Deep learning regression network | High accuracy out-of-the-box | Requires GPU, slower initialization |
| MOSSE | Minimum Output Sum of Squared Error | Extremely fast | Lower accuracy and drift |

## Logical steps in code

1. **Asset Download**
   * Download the video asset ZIP using urlretrieve, extract with ZipFile.
2. **Tracker Selection**
   * Define tracker\_types list and choose one (e.g., KCF).
   * Instantiate via the corresponding cv2.TrackerXXX.create() or cv2.legacy.TrackerXXX.create() call depending on API.
3. **Video I/O Setup**
   * Open input video with cv2.VideoCapture.
   * Retrieve frame width and height.
   * Initialize cv2.VideoWriter to save output frames (XVID codec at 10 FPS).
4. **Define Initial ROI**
   * Hardcode bbox (e.g., (1300, 405, 160, 120)), or allow interactive selection via cv2.selectROI.
   * Visualize ROI on the first frame with drawRectangle and plt.imshow.
5. **Tracker Initialization**
   * Call tracker.init(frame, bbox) on the first frame to set up the appearance model and initial motion state.
6. **Frame-by-Frame Tracking Loop**
   * Read each new frame.
   * Start timer (cv2.getTickCount()).
   * Call tracker.update(frame) to predict and locate the object, returning ok and updated bbox.
   * Compute FPS from elapsed ticks.
   * Draw rectangle if ok; else display “Tracking failure.”
   * Overlay tracker type and FPS text via cv2.putText.
   * Write annotated frame to output video.
7. **Post-Processing**
   * Release VideoCapture and VideoWriter.
   * (Optional) Re-encode the saved video to H.264 via ffmpeg for better compatibility.

# Face Detection

In real‑time applications, automatically locating human faces is a crucial first step for tasks such as alignment, recognition, or anonymization. Section 6 describes how to load a pre‑trained Single Shot Multibox Detector (SSD) model in Caffe format, preprocess video frames into DNN blobs, run inference, and annotate each detection with bounding boxes and confidence scores.

## Model Setup & Frame Preprocessing

Network: A Caffe SSD trained on face data (deploy.prototxt + .caffemodel).

Blob conversion: Resize each frame to 300×300, subtract mean values [104,117,123], and pack into a 4D tensor via cv2.dnn.blobFromImage(frame, 1.0, (300,300), mean).

Input assignment: net.setInput(blob) readies the network for a forward pass.

## Inference & Detection Filtering

Forward pass: detections = net.forward() returns detections of shape [1×1×N×7], where each entry contains [confidence, x1, y1, x2, y2].

Thresholding: Discard boxes with confidence < 0.7 to reduce false positives.

Coordinate mapping: Multiply normalized output coordinates by the original frame width/height to recover pixel values.

## Annotation & Performance Reporting

Drawing boxes: For each valid detection, draw a green rectangle via cv2.rectangle(frame, top\_left, bottom\_right, (0,255,0)).

Overlay labels: Render the confidence score above each box using cv2.putText.

Timing: Retrieve inference time with t, \_ = net.getPerfProfile() and convert ticks to milliseconds, displaying it at the top of the frame.

By following these steps, you achieve a lightweight, dependency‑free face detector capable of processing live camera feeds at interactive frame rates, while providing visually annotated feedback for every detected face.

# 7. Deep Learning based Object Detection

## 7.1Architecture and Model

In this section we employ a Single Shot Multi-Box Detector (SSD) built atop the lightweight MobileNetV2 backbone. SSD formulates object detection as a single feed-forward convolutional network that simultaneously predicts class confidences and bounding-box offsets at multiple feature-map scales. MobileNetV2 provides a compact, depth-wise separable convolutional base that offers real-time throughput on CPU or modest GPUs, making it ideal for embedded or desktop applications. The pre‐trained COCO‐trained “ssd\_mobilenet\_v2\_coco\_2018\_03\_29” model includes a frozen inference graph (frozen\_inference\_graph.pb) and a textual graph definition (.pbtxt), encapsulating the learned feature extractors, prior box definitions, and detection heads.

## 7.2 Model Setup and Configuration

We start by loading the pre-trained SSD model into OpenCV’s DNN module using cv2.dnn.readNetFromTensorflow(), giving it the paths to the frozen inference graph (.pb) and its accompanying text config (.pbtxt). Next, we read our coco\_class\_labels.txt file into a Python list so that each numeric class ID can be shown as a real object name (e.g., “person,” “bicycle,” “dog,” etc.).

When it’s time to run detection on an image, we turn that image into a “blob,” which simply means:

1. **Resize** the image to 300×300 (the network’s expected input size),
2. **Subtract** a fixed mean value from each channel (to normalize brightness), and
3. **Reorder** the color channels if needed (BGR → RGB).

This blob is then passed straight into the network, guaranteeing that every image—no matter its original size or color balance—arrives in exactly the format the model was trained on.  
  
Instructions for Manual Setup:

Download Model files from Tensorflow model ZOO

Model files can be downloaded from the Tensorflow Object Detection Model Zoo: [tf2\_detection\_zoo.md](https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md)

Download mobilenet model file

You can download the [model TAR.GZ file](http://download.tensorflow.org/models/object_detection/ssd_mobilenet_v2_coco_2018_03_29.tar.gz) and uncompress it.

After Uncompressing and put the highlighed file (along with the folder) in a models folder.

**ssd\_mobilenet\_v2\_coco\_2018\_03\_29**  
|─ checkpoint  
|─ **frozen\_inference\_graph.pb**  
|─ model.ckpt.data-00000-of-00001  
|─ model.ckpt.index  
|─ model.ckpt.meta  
|─ pipeline.config  
|─ saved\_model  
|─── saved\_model.pb  
|─── variables

Create config file from frozen graph

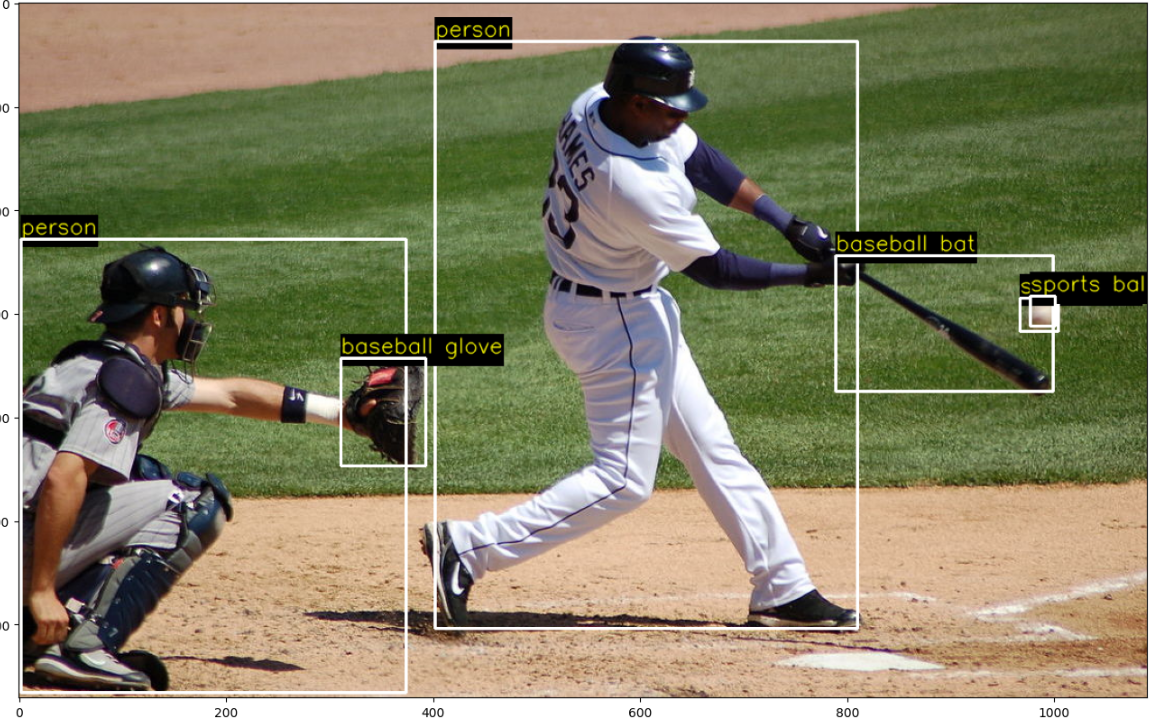
1. Extract the files
2. Run the [tf\_text\_graph\_ssd.py](https://github.com/opencv/opencv/blob/master/samples/dnn/tf_text_graph_ssd.py) file with input as the path to the frozen\_graph.pb file and output as desired.

**A sample config file has been included in the models folder**

## Inference Pipeline

The detection pipeline unfolds in two main stages:

### Forward Pass and Object Extraction

Once the blob is set via net.setInput(), a single call to net.forward() produces a three-dimensional array of detection records. Each record encodes [batch\_id, class\_id, confidence, x\_min, y\_min, x\_max, y\_max], with the box coordinates normalized to [0,1]. We iterate over all detections, filter out low-confidence entries below a chosen threshold (e.g., 0.25), and rescale the normalized coordinates back to pixel values by multiplying by the image’s width and height.  


### Visualization and Performance Metrics

For each retained detection we draw a white rectangle and superimpose the class label in yellow text. To ensure legibility against varying backgrounds, we compute the text’s pixel dimensions via cv2.getTextSize(), draw a filled black rectangle as its backdrop, then render the label with cv2.putText(). After processing each frame, we also retrieve the network’s performance profile (net.getPerfProfile()) to compute and display the average inference time in milliseconds. Converting the BGR result to RGB enables us to visualize final outputs inline using Matplotlib.

This streamlined, paragraph‐based presentation of the SSD–MobileNet pipeline integrates both the theoretical underpinnings of one-stage detection and the practical OpenCV code patterns needed to load, run, and display deep‐learning–based object detection results.

# Pose Estimation using Open Pose

OpenPose is a landmark real-time system that recovers 2D body poses for multiple people in a single pass. It does so by simultaneously predicting **where** each joint is (via confidence maps) and **how** joints connect into limbs (via Part Affinity Fields). This end-to-end approach requires no person-by-person cropping or post hoc tracking, making it both fast and robust for crowded scenes.

## 8.1 Theory of Part Affinity Fields

At its core, OpenPose casts pose estimation as a **joint graph inference** problem.

1. **Confidence Maps**: For each of the n body keypoints (e.g., nose, shoulders, elbows…), the network outputs a 2D heat-map Ci(x,y). High values of Ci indicate pixels likely to contain joint i.
2. **Part Affinity Fields (PAFs)**: For each of the mmm limb types (e.g., upper arm, lower leg), the network produces a 2-channel vector field Fj(x,y) = (Fx\_j, Fy\_j). At each pixel, Fj points from one joint in the pair toward the other, encoding both orientation and association strength.

Once these maps are predicted, OpenPose constructs a graph whose nodes are candidate joint locations (peaks in the confidence maps) and whose edges are scored by integrating the PAF vectors along the line segment between two joints. A **bipartite matching** (for each limb type) then finds the best connections, and a final greedy grouping step produces disjoint skeletons for each person.

## 8.2 Model Architecture

We use the “line-vector” variant of OpenPose, implemented as a single deep network with two parallel output branches:

* A stack of convolutional layers that refines and upsamples intermediate features into n confidence maps.
* A sibling stack that outputs 2m2m2m PAF channels.

In our setup:

* n=15 keypoints (MPII standard)
* m=14 limbs (neck→shoulder, shoulder→elbow, etc.)
* Input resolution: **368×368** (trade-off between localization accuracy and throughput)
* Backbone: VGG-style feature extractor followed by specialized layers for each output branch.

## 8.3 Inference Pipeline

Blob Creation:  
blob = cv2.dnn.blobFromImage(

image, 1/255.0, (368,368),

mean=(0,0,0), swapRB=True, crop=False

)

net.setInput(blob)

output = net.forward()

🡪🡪 This produces a tensor of shape (1,n+2m,H,W).

Keypoint Detection:  
For each confidence map Ci, find the global maximum via cv2.minMaxLoc(). If its value ≥ threshold (e.g., 0.1), record its (x,y) after scaling from (W,H) back to the original image size.

PAF-Based Matching:  
For each limb j connecting keypoints (a,b):

* Sample several points along the line segment between every candidate of a and b.
* At each sample, dot the PAF vector Fj with the normalized candidate direction.
* Average these dot products to score the edge.
* Run a bipartite matcher to select the highest-scoring pairs without conflicts.

Skeleton Assembly:

Link the matched joints into separate person instances. Discard any partial skeletons with too few joints.

## 8.4 Constraints and Performance

* **Real-Time Requirement**: Operating at ≥10 FPS on a modern CPU requires limiting input resolution and network depth.
* **Crowd Robustness**: PAF matching scales roughly as O(k2m) where k is keypoint candidates per type; high crowd density can slow matching.
* **Threshold Sensitivity**: Low confidence thresholds may yield false positives; high thresholds can drop valid joints.
* **Occlusion Handling**: OpenPose can recover partially occluded limbs if at least one end remains visible and PAFs remain strong.

Balancing these factors is key: larger input sizes and deeper networks boost accuracy but cut throughput, while sparser detection candidates speed matching at the risk of missing small or distant people.

# Conclusion

Throughout this manual, we have explored the breadth of OpenCV’s core functionality—from basic image I/O and pixel‐level manipulations to advanced computer‐vision pipelines powered by deep neural networks. You’ve seen how to:

* **Load, display, and interrogate** images and video streams.
* **Annotate** visuals with lines, circles, rectangles, and text.
* **Enhance** image quality via brightness, contrast, thresholding, and bitwise operations.
* **Detect** and **match** features for image alignment, panorama stitching, and HDR merging.
* **Track** moving objects using classical algorithms (MeanShift, KCF, CSRT, MOSSE, etc.).
* **Locate** faces in real time with a lightweight SSD–Caffe model.
* **Perform** deep‐learning object detection (SSD–MobileNet) and multi‐person pose estimation (OpenPose) entirely within OpenCV’s DNN module.