Free Fall and Influence of Air Resistance

Andrejs Cvečkovskis, Rauls Poļs, Jēkabs Priedītis

Abstract

In this work, a video-based experimental study of free fall and the influence of air resistance is presented. Video recordings of a falling object are analysed to reconstruct its vertical motion and extract kinematic quantities such as velocity and acceleration. Image processing techniques are employed to isolate the object from the background and track its position over time in an automated manner. Different processing approaches are compared in terms of stability and consistency of the reconstructed trajectory. The obtained motion data are used to examine the agreement between experimental observations and theoretical models of free fall with and without air resistance. The study shows both the potential and the limitations of video-based motion analysis.

1 Theoretical Background

When an object falls through the air, its motion is governed by the balance between gravitational force mg and quadratic drag force bv^2. According to Newton's second law, the equation of motion can be written as

$$ma = mg - bv^2 \tag{1}$$

where m is the mass of the object, a is the instantaneous acceleration, g is the gravitational acceleration, v is the instantaneous velocity and b is the drag coefficient, which can be expressed as

$$b = \frac{m(g-a)}{v^2} \tag{2}$$

This can be solved by numerical methods, for example, by using Euler's method.

The analytical solution of the equation (1) for the displacement of the object is in the following form:

$$y(t) = \frac{mg}{b} \ln\left(\cosh\left(t\sqrt{\frac{gb}{m}}\right)\right) \tag{3}$$

However, assuming no drag, the equation of motion simplifies considerably to $y(t) = gt^2/2$, implying all objects should fall at the same time from the same height.

2 Experimental Setup

The experiment consisted of recording the free fall of various objects using video analysis. Each object was dropped from an approximate height of 2.5 m against a uniform yellow wall, chosen to maximise contrast and facilitate image segmentation. A description of all the objects analysed is provided in Table 1.

Object	**Mass (g)**
Plastic sphere	61
Polystyrene sphere	1.83
Small plastic disk	14
Medium-sized plastic disk	27
Large plastic disk	61
Half-sphere	21
Plastic bag	18

Table 1: Objects used in the experiment with their masses.

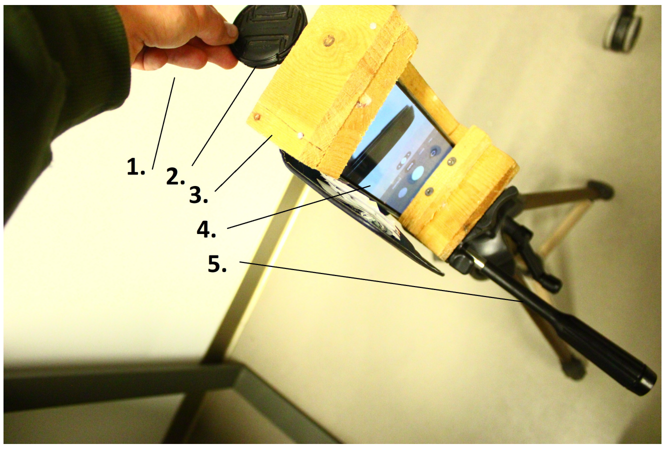

Figure 1: The experimental setup. 1. hand, 2. object, 3. custom-made phone holder for camera tripod, 4. phone, 5. camera tripod

3 Image Processing

This section mostly includes the description of the applied methods in semester project.py. The methods used in 10122025v.3.0.py are mentioned separately. The experimental data consist of a video recording of a freely falling object against a visually simple background. All image processing and trajectory reconstruction steps are performed automatically using a custom Python pipeline. The purpose of the image processing stage is to extract a reliable one-dimensional vertical trajectory $y(t)$ from the video frames while minimising user intervention and subjective bias [GW08, Sze10].

3.1 Motion Onset Detection

The analysis begins by detecting the onset of motion in the video. Consecutive grayscale frames are compared using absolute pixel differences, and the fraction of pixels exceeding a fixed threshold is evaluated. Motion is declared when this fraction exceeds a predefined limit for several consecutive frames [Pic04].

This procedure allows the algorithm to identify the release of the object and discard initial frames containing only static background.

3.2 Background Estimation and Region of Interest Selection

A static background image is estimated as the pixel-wise median of several frames preceding the detected motion onset. Median-based background estimation is known to be robust against noise and transient foreground objects [Pic04].

To reduce noise and computational cost, the analysis is restricted to an automatically selected region of interest (ROI). The ROI is determined by detecting foreground pixels in the first frames after motion onset and expanding the bounding box of detected motion by a fixed margin. This ensures that the ROI contains the entire object trajectory while excluding irrelevant background regions [Sze10].

3.3 Foreground Segmentation

Three independent segmentation methods are applied within the ROI:

- adaptive thresholding,
- background subtraction,
- a combined method using the logical intersection of the two.

Adaptive thresholding and background subtraction are standard techniques for isolating moving objects under non-uniform illumination conditions [GW08, Pic04]. Each segmentation mask is post-processed using median filtering and morphological opening and closing operations to suppress noise and remove small artefacts [GW08]. Pixels near the top of the ROI are explicitly masked to prevent interference from the hand releasing the object.

3.4 Contour Selection and Object Tracking

For each segmentation method, object tracking is performed frame by frame using contour analysis. Candidate contours are filtered based on minimum area and circularity criteria, ensuring that only compact, approximately circular objects are considered [Bra00].

When multiple candidate contours are present, temporal continuity is enforced by selecting the contour whose centroid lies closest to the previously detected object position. Large frame-to-frame jumps are rejected, and short gaps in detection are tolerated up to a fixed limit. Such continuity-based contour selection is a standard strategy in object tracking to suppress spurious detections [YJS06].

The object position in each frame is defined as the centroid of the selected contour. Tracking is terminated if the object cannot be detected reliably for several consecutive frames.

3.5 Method Selection

Each segmentation method produces a candidate trajectory in pixel coordinates. To identify the most reliable method, a scoring procedure is applied based on:

- the length of the longest approximately monotonic segment,
- the root-mean-square residual of a quadratic fit,
- the roughness of the second finite difference,
- the fraction of frames with valid detections.

This multi-criteria evaluation allows a quantitative comparison of segmentation robustness and temporal consistency without manual intervention [Sze10].

3.6 Physics Interval and Impact Detection

From the selected trajectory, the longest monotonic downward segment is extracted and interpreted as the physically valid free-fall interval. The impact point is detected using a combination of criteria:

- abrupt changes in velocity,

- sustained velocity plateaus near the bottom of the ROI,
- loss of reliable tracking close to the lower boundary of the ROI.

Restricting the analysis to the physically meaningful interval is essential to avoid contamination from post-impact dynamics and tracking artefacts [Bei96, Bro13].

3.7 Pixel-to-Meter Calibration

The vertical pixel coordinate is converted to physical units using the known drop height $H = 2.500\,\mathrm{m}$. The meter-per-pixel scale is computed as

$$\alpha = \frac{H}{y_{\text{impact}} - y_0},$$

where y_0 is the median initial vertical position and y_{impact} is the detected impact position in pixel coordinates.

This calibration approach is commonly used in video-based kinematic experiments when explicit metric references are limited [Bei96]. If impact detection fails or the inferred displacement is inconsistent with the ROI geometry, the analysis proceeds in pixel units, and a warning is issued.

3.8 Trajectory Smoothing and Derivatives

The calibrated vertical trajectory $y(t)$ is smoothed using a cubic smoothing spline. Velocity and acceleration are obtained analytically by differentiating the spline, yielding $v(t)$ and $a(t)$ without numerical differencing.

Spline-based smoothing provides stable derivative estimates in the presence of measurement noise and is widely used for reconstructing kinematic quantities from experimental data [Rei67, Wah90]. The spline smoothing parameter is chosen to suppress pixel-scale noise while preserving the overall kinematic structure of the motion.

3.9 Output Products

The image processing pipeline produces the following outputs:

- diagnostic images showing raw frames, segmentation masks, and detected contours,
- the reconstructed vertical trajectory $y(t)$,
- velocity and acceleration as functions of time, $v(t)$ and $a(t)$,
- phase-space representations $v(y)$ and $a(y)$,
- quantitative diagnostics comparing segmentation methods.

These results form the basis for the physical interpretation presented in the subsequent sections.

3.10 Methods of 10122025 v3.0.py

The methods used in 10122025 v3.0.py:

- Grayscale conversion - The original video frames are recorded in colour (BGR format). Since colour information is not required for object detection in this experiment, the frames are converted to grayscale. This reduces the data dimensionality from three channels to one and simplifies subsequent processing steps, particularly thresholding.
- Global thresholding converts the grayscale image into a binary image by applying a single intensity threshold to all pixels. Pixels darker than the threshold are classified as foreground (the falling object), while brighter pixels are classified as background. The inverted binary mode is used so that the object appears as a white region on a black background.

  This method is effective because the object has a consistent intensity contrast relative to the background, and the lighting conditions remain stable throughout the video.
- Morphological opening - Morphological opening removes small isolated noise elements and spurious pixels from the binary image. This step suppresses high-frequency noise while preserving the overall shape of the object.
- Morphological closing - Morphological closing fills small holes and gaps within the detected object region and ensures that the object is represented as a single, compact shape. This improves the robustness of subsequent contour detection.
- Contour detection - Contours are extracted from the cleaned binary mask. Only external contours are retrieved, which avoids detecting internal structures and reduces computational complexity. Each contour corresponds to a connected foreground region in the image.
- Largest contour - The falling object is assumed to be the largest connected foreground object in the frame. Selecting the contour with the largest area effectively removes residual

noise and ensures that only the target object is tracked.

- Centroid calculation by using image moments - Image moments are used to compute the centroid (centre of mass) of the detected object. The vertical coordinate of the centroid provides a robust and smooth estimate of the object's position, which is less sensitive to shape deformations than edge-based measurements.

- Pixel-to-physical unit conversion - The centroid position measured in pixels is converted into physical units (meters) using a known reference distance. This allows the experimentally obtained trajectory to be directly compared with theoretical models of free fall and air resistance.

Unlike semester project.py, the videos were not cut automatically; therefore first/last frame of motion had to be selected manually, with min-bound and max-bound values. The code also provides manual selection of thresholding values as well as kernel size to correctly plot the motion trajectory. The distance was not automatically determined, but pre-selected, meaning that the release of objects had to be coordinated extremely carefully to acquire a physically reasonable comparison with the theoretical models.

4 Results and Discussion

4.1 Results of `10122025 v3.0.py`

For each object, 5 trajectory measurements were conducted, all of which were successfully acquired. Trajectories of sphere, hemisphere and disk models were acquired and graphed using 10122025 v3.0.py by adjusting threshold values and the selection of first/last frame of the motion. The foam plastic and plastic bag models were acquired and graphed using semester project.py. Despite the existence of successful measurements, the results below show an inaccuracy of some degree. The discussion will highlight the prominent causes of these inaccuracies, which are inevitable in the presence of humans, and to a smaller degree, phones as well. The free-fall acceleration was calculated with the Euler numerical approach. The spherical models were deemed useful for this calculation as a sphere should have the least drag coefficient value, and drag force would influence the trajectory only after a long time of free-fall, as confirmed in Fig. 3. After 5 acquired free-falling sphere trajectories, the mean g value was calculated to be approximately

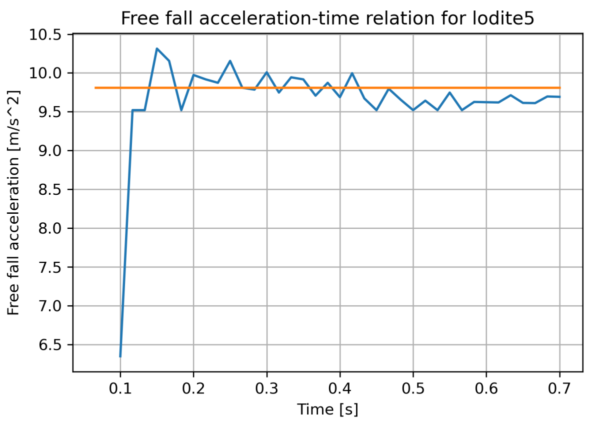

Figure 2: Free fall acceleration-time relation for a free-falling sphere

9.7 m/s^2, which is close to the actual value. From now on, the actual value will be used in the free-fall trajectory calculations. The acquired sphere trajectories (see Fig. 3) matched both the free-fall and quadratic drag model, indicating our assumption of nearly drag-less motion is justified.

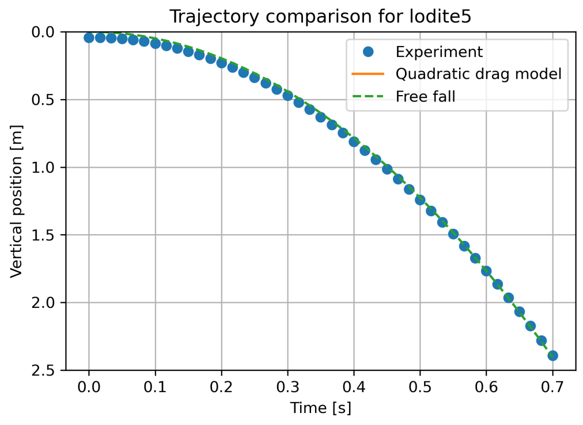

Figure 3: The trajectory of a sphere compared with theoretical models.

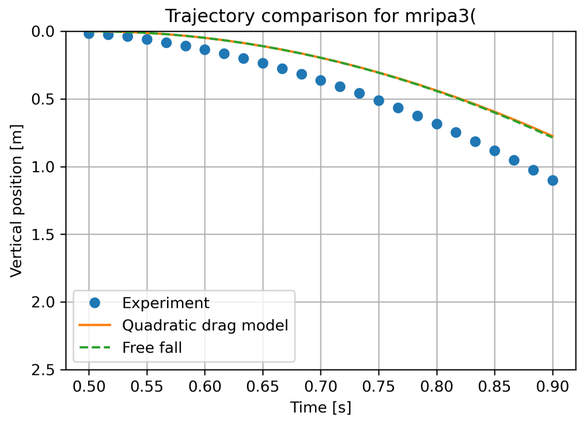

Figure 4: The trajectory of the small disk vs theoretical models

On the other hand, the small disk trajectories (see Fig. 4) were positioned below free-fall and drag models (which matched), indicating the small disk motion was more rapid than expected. This could be attributed to the arbitrary position of the object at the beginning of time, which was selected manually by hand. As the result, the match of both theoretical models could also be attributed to poor numerical calculations. Similarly, the offset of the

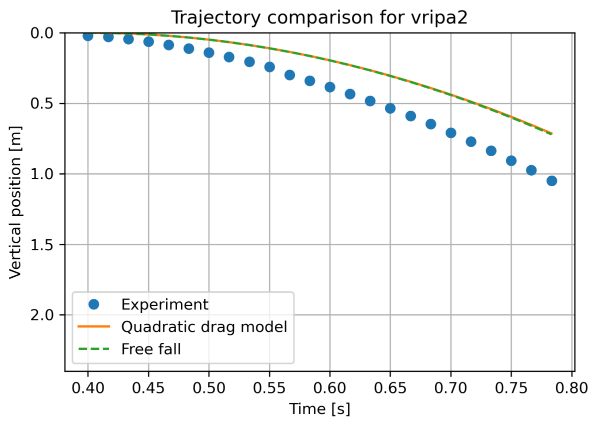

Figure 5: The trajectory of the medium-sized disk vs theoretical models

medium-sized disk (see Fig. 5) could be explained with the same effects. On the other hand, the suc-

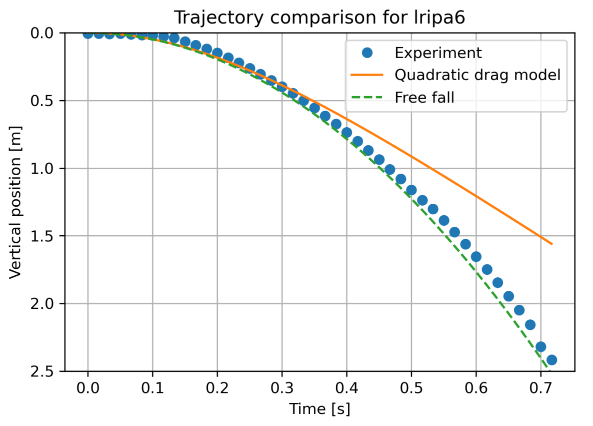

Figure 6: The trajectory of the large disk vs theoretical models

cessful initial positioning of the large disk (Fig. 5) resulted in acquiring an experimental model that is positioned between the theoretical free-fall and the drag models, where the difference between drag and the experimental model could be attributed to initial time conditions, which could be improved by choosing a better frame rate. The hemisphere model has a dual nature: the theoretical models of a falling hemisphere with spherical side downwards (Fig. 7) show traits similar to those of the sphere, whereas the case of spherical side upwards

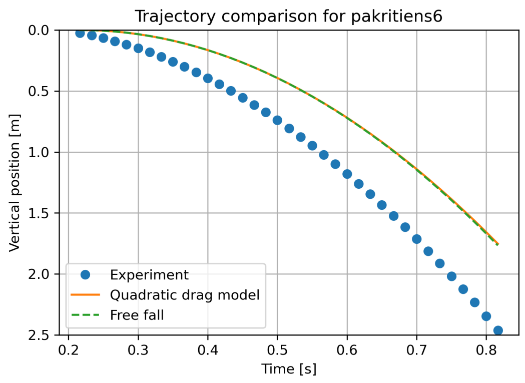

Figure 7: The trajectory of the hemisphere (spherical side downwards) vs theoretical models)

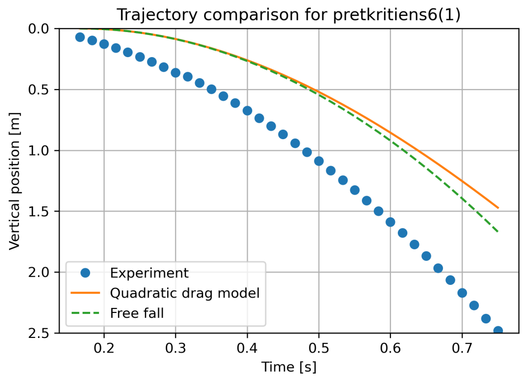

Figure 8: The trajectory of the hemisphere (spherical side upwards) vs theoretical models

(Fig. 8) shows a slightly different nature, with theoretical models not matching, as they should. The experimental model offsets between the experimental models could be attributed to the arbitrary selection of the initial position.

4.2 Results of `semester project.py`

The reconstructed vertical trajectories are presented for two representative objects: a plastic ball and a lightweight bag (maisiņš). These cases were selected to show the effect of object geometry and air resistance on the observed free-fall dynamics, as obtained from the image processing script.

4.2.1 Plastic ball

The vertical trajectory of the plastic ball, shown in Fig. 9, exhibits a smooth and approximately quadratic dependence on time. Within the physically valid interval identified by the algorithm, the experimental data closely follow the theoretical free-fall prediction. Only minor deviations are observed near the lower end of the trajectory, where tracking quality decreases due to proximity to the impact region.

The good agreement with the ideal free-fall model is consistent with the compact shape and relatively small aerodynamic drag of the plastic ball (see Fig. 10). This result indicates that, for sufficiently rigid and symmetric objects, the implemented image processing and trajectory reconstruction methods are capable of recovering physically realistic motion with minimal systematic bias.

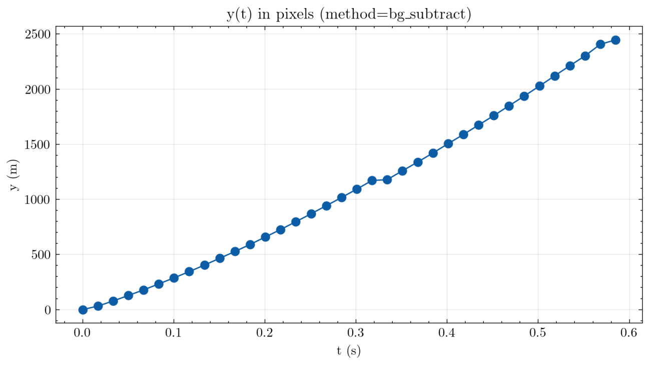

Figure 9: Reconstructed vertical trajectory $y(t)$ of the plastic ball compared to theoretical free-fall motion.

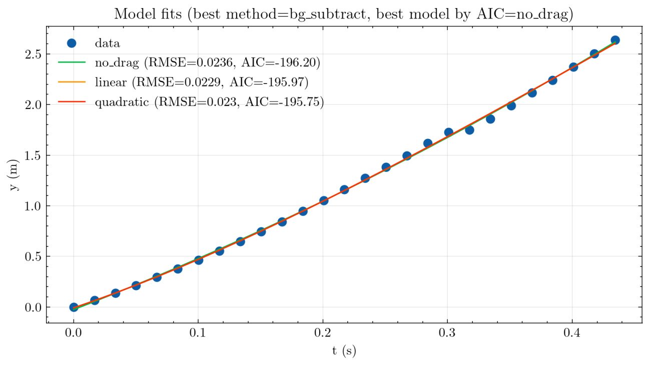

Figure 10: Reconstructed vertical trajectory $y(t)$ of the plastic ball compared to theoretical models with drag.

4.2.2 Bag (maisiņš)

The reconstructed trajectory of the bag is shown in Fig. 11. In contrast to the plastic ball, the motion deviates noticeably from the ideal quadratic behaviour expected for free fall. The descent occurs more slowly, and the trajectory curvature indicates a significant influence of air resistance throughout the observed interval.

These deviations are physically expected due to the large effective cross-sectional area and irregular shape of the bag, which result in increased drag forces (see Fig. 12). Consequently, the ideal free-fall model systematically overestimates the fall speed, while models including air resistance provide a more appropriate qualitative description.

The trajectory of the bag also exhibits increased scatter compared to the plastic ball. This can be attributed both to physical effects, such as deformation and oscillatory motion, and to limitations of the image processing stage, where changing object shape complicates reliable contour detection.

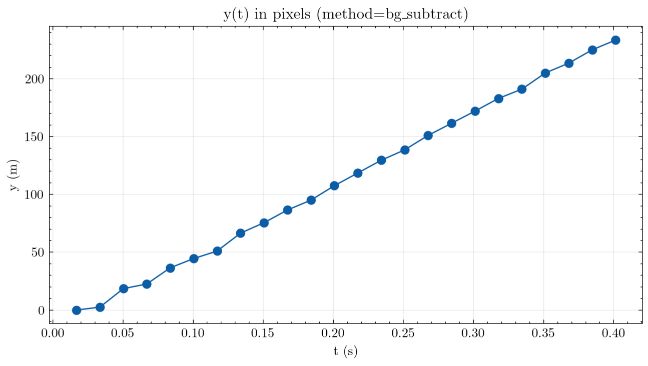

Figure 11: Reconstructed vertical trajectory $y(t)$ of the bag (maisiņš), illustrating strong deviations from ideal free-fall motion due to air resistance.

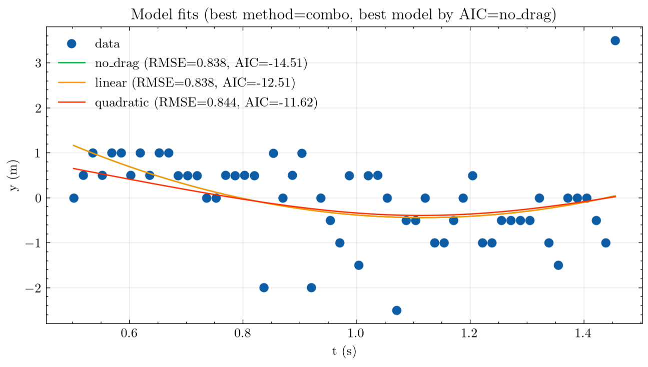

Figure 12: Reconstructed vertical trajectory $y(t)$ of the bag (maisiņš) compared to theoretical models with drag, illustrating strong deviations from ideal free-fall motion due to air resistance.

4.2.3 Discussion and limitations

The comparison between the plastic ball and the bag demonstrates that the reconstructed trajectories are sensitive to differences in aerodynamic properties. While the plastic ball follows motion close to ideal free fall, the bag clearly exhibits drag-dominated behaviour.

The main sources of uncertainty in the results arise from the finite frame rate of the recording, pixel-level localisation noise, and the need to smooth the measured trajectory prior to differentiation. These limitations are particularly relevant for non-rigid objects such as the bag, where shape changes introduce additional variability.

Despite these constraints, the qualitative differences between the two cases are robust and physically consistent. The results confirm that the applied image processing approach is sufficient to distinguish between near-ideal free fall and motion strongly influenced by air resistance using standard video recordings.

5 Conclusions

The results demonstrate how image processing techniques can be used to extract quantitative physical information from video data. Differences in fall behaviour between objects of varying mass and shape highlight the role of air resistance and examine the suitability of simple video analysis for studying non-ideal free fall.

The analysis further shows that the reliability of the extracted physical quantities strongly depends on the quality of the recorded footage. In the present experiment, the limited frame rate of the recordings, as well as image quality and the shade of the object, were identified as the most substantial sources of error.

In order to improve the results of the experiment, a camera with a larger frame rate and better quality could be utilised. Additionally, the object could be dropped from a larger height and in front of a more uniform background to minimise errors. The objects could be dropped by an automated system instead of a hand to minimise object rotational effects and their influence on free fall, as well as inaccuracies due to the arbitrary choice of initial positions. Nevertheless, despite these limitations, image processing provided sufficiently accurate trajectory data to capture the qualitative trends predicted by theoretical models.

6 Final notes

Contributions to the project: Andrejs Cvečkovskis - coding (semester project.py), report, presentation (text)

Rauls Poļs - coding (10122025 v3.0.py), experimental setup, report, presentation (results)

Jēkabs Priedītis - experimental setup, report and presentation (structure, text)

References

[Bei96] Robert J. Beichner. The impact of video motion analysis on kinematics graph interpretation skills. *American Journal of Physics*, 64:1272–1277, 1996.

[Bra00] Gary Bradski. The opencv library. *Dr. Dobb's Journal of Software Tools*, 2000.

[Bro13] Douglas Brown. Video analysis for introductory physics: A review. *The Physics Teacher*, 51:458–460, 2013.

[Cha11] Rick Chartrand. Numerical differentiation of noisy, nonsmooth data. *ISRN Applied Mathematics*, 2011.

[GW08] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing.* Pearson, 3 edition, 2008.

[Kal60] R. E. Kalman. *A New Approach to Linear Filtering and Prediction Problems*, volume 82. 1960.

[Pic04] Massimo Piccardi. Background subtraction techniques: a review. *IEEE International Conference on Systems, Man and Cybernetics*, pages 3099–3104, 2004.

[Rei67] C. H. Reinsch. Smoothing by spline functions. *Numerische Mathematik*, 10:177–183, 1967.

[SG99] Chris Stauffer and W. E. L. Grimson. Adaptive background mixture models for real-time tracking. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 246–252, 1999.

[Sze10] Richard Szeliski. *Computer Vision: Algorithms and Applications.* Springer, 2010.

[Wah90] Grace Wahba. *Spline Models for Observational Data.* SIAM, 1990.

[YJS06] Alper Yilmaz, Omar Javed, and Mubarak Shah. *Object Tracking: A Survey*, volume 38. 2006.

Appendix A: 10122025 v3.0.py

```
import cv2
import numpy as np
import os
import matplotlib.pyplot as plt
from scipy.signal import savgol_filter


def path_quadratic(m, g, b2, t):
    vT = np.sqrt(m * g / b2)
    return (vT**2 / g) * np.log(np.cosh(g * (t-t[0]) / vT))


def video(m, g, min_bound, max_bound, fname, distance,thrmin,thrmax,kersize):

    cap = cv2.VideoCapture(fname)
    namestr = os.path.splitext(os.path.basename(fname))[0]

    frames = []
    centers_y_meters = []
    times = []

    frame_id = 0
    fps = 60

    while True:
        ret, frame = cap.read()
        if not ret:
            break
#Image processing:
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        _, mask = cv2.threshold(gray, thrmin, thrmax, cv2.THRESH_BINARY_INV)

        kernel = np.ones((kersize, kersize), np.uint8)
        mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)
        mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)

        contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL,
                                       cv2.CHAIN_APPROX_SIMPLE)
        if len(contours) == 0:
            frame_id += 1
            continue

        cnt = max(contours, key=cv2.contourArea)
        M = cv2.moments(cnt)
        if M["m00"] == 0:
            frame_id += 1
            continue

        cy = int(M["m01"] / M["m00"])

        image_height_px = frame.shape[0]
        meters_per_pixel = distance / image_height_px
        y_meters = cy * meters_per_pixel

        frames.append(frame_id)
        centers_y_meters.append(y_meters)
        times.append(frame_id / fps)

        frame_id += 1
```

```
#---------------------
#numerical part
    cap.release()
    t = np.array(times[min_bound:max_bound])
    y = np.array(centers_y_meters[min_bound:max_bound])

    #y = savgol_filter(y, 11, 3) unused, bad

    #Numerical Euler mahincijas
    dt = t[1] - t[0]

    v = (y[1:] - y[:-1]) / dt
    a = (v[1:] - v[:-1]) / dt

    t_mid = t[2:]

    # ----------------------------
    #b2 from data
    # ----------------------------
    b2_list = []

    for k in range(1, 10): #take only the last frames (might have to change the range upper
        value in case of low frame count)
        v_tail = v[-k:]
        a_tail = a[-k:]

        mask = v_tail > 0.1
        if np.sum(mask) < 3:
            continue

        b2_values = m * (g - a_tail[mask]) / (v_tail[mask]**2)
        b2_est = np.mean(b2_values)

        b2_list.append(b2_est)

    b2_mean = np.mean(b2_list)

    print(f"Estimated␣quadratic␣drag␣coefficient␣b2␣=␣{b2_mean:.4e}␣kg/m")

    plt.figure()
    plt.plot(t, y, "o", label="Experiment")

    plt.plot(t,
             path_quadratic(m, g, b2_mean, t),
             label="Quadratic␣drag␣model")

    plt.plot(t,
             0.5 * g * (t-t[0])**2,
             "--",
             label="Free␣fall")

    plt.xlabel("Time␣[s]")
    plt.ylabel("Vertical␣position␣[m]")
    plt.title("Trajectory␣comparison␣for␣{}".format(namestr[:-3])) #might have to tamper with
        .format(namestr...) if the title is not looking good
    plt.legend()
    plt.grid(True)

    plt.ylim(distance, 0)
    plt.savefig("{}_quadratic_drag.png".format(namestr[:-2]), dpi=300)
    plt.show()
    return b2_mean


#object--mass[kg]
#normal ball: 0.045
#the smaller disk: 0.014
#medium disk: 0.027
#the bigger disk: 0.061
#hemisphere: 0.021
#maisi: 0.018
#foam plastic ball 0.00183




#video(0.021,9.81,0,-5,"C:/Users/raulss-/Documents/pretkritiens4.mp4",2.3)
#video(0.027,9.81,24,-16,"vripa2(2).mp4",2.4,130,255,15)


#video(0.061,9.81,0,-17,"lripa6.mp4",2.5,130,255,15)


#video(0.027,9.81,30,-22,"mripa3(1).mp4",2.5,100,255,5)

video(0.021,9.81,10,-5,"pretkritiens6(1).mp4",2.5,120,255,7)
```

Appendix B: semester project.py

```
#!/usr/bin/env python3
"""
semester_project.py
Auto-detect␣meters-per-pixel␣from:
␣␣known␣drop␣height␣H␣=␣2.5␣m
␣␣+␣detected␣start-of-fall␣y_px
␣␣+␣detected␣impact␣y_px␣(stop/bounce/plateau)

Pipeline:
1)␣Load␣video
2)␣Detect␣motion␣start,␣skip␣release␣frames
3)␣Build␣median␣background
4)␣Auto␣ROI␣crop␣around␣motion␣corridor
5)␣Track␣multiple␣segmentation␣methods␣with␣temporal␣continuity␣+␣circle␣center
6)␣Choose␣best␣tracking␣method␣by␣score
7)␣Using␣best␣track:␣detect␣impact␣(pixel␣domain),␣compute␣meters_per_pixel␣=␣H␣/␣(y_imp␣-␣y0)
8)␣Convert␣y(t)␣to␣meters,␣refit␣physics␣models,␣save␣plots␣+␣summary
"""

from __future__ import annotations

import os
import math
import json
import argparse
from dataclasses import dataclass
from typing import Dict, List, Optional, Tuple

import cv2
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import least_squares
from scipy.interpolate import UnivariateSpline

# OPTIONAL PLOT STYLE #

import scienceplots
plt.style.use(['science'])


# -----------------------------
# Config defaults
# -----------------------------

DEFAULT_DROP_HEIGHT_M = 2.5 # <- your height

# Motion detection
MOTION_DIFF_THR = 15
MOTION_FRAC_THR = 0.002
MOTION_CONSEC = 3
MAX_SEARCH_SECONDS = 6.0

# Background
BG_SAMPLES = 30

# ROI
ROI_LOOK_FRAMES = 25
ROI_BG_THR = 25
ROI_MARGIN = 40
```

```
# Segmentation + cleanup
TOP_CUTOFF_PX = 80 # you said hand is at top -> suppress it
GAUSS_BLUR = (7, 7)
ADAPTIVE_BLOCK = 31 # must be odd
ADAPTIVE_C = 7
BG_DIFF_THR = 25
OPEN_K = 5
CLOSE_K = 7

# Tracking filters
MIN_AREA_PX = 80
MIN_CIRC = 0.15
MAX_JUMP_PX = 120
MAX_MISSES = 6

# Physics interval
MIN_POINTS_PHYS = 8
MONO_TOL_PX = 3.0

# Kinematics / smoothing
SPLINE_S = 2e-4

# Model fitting
G_INIT = 9.81

# Method scoring weights
LAMBDA_COV = 0.02
MU_ROUGH = 1.0


# ----------------------------
# Data structures
# ----------------------------

@dataclass
class TrackResult:
    method: str
    fps: float
    roi: Tuple[int, int, int, int] # x0,y0,x1,y1 in full-frame coords
    t: np.ndarray # seconds, starting at analysis_start frame
    x_px: np.ndarray # ROI coords
    y_px: np.ndarray # ROI coords (down = +)
    valid: np.ndarray # bool

    @property
    def coverage(self) -> float:
        return float(np.mean(self.valid)) if len(self.valid) else 0.0


# ----------------------------
# Utilities
# ----------------------------

def ensure_dir(p: str) -> None:
    os.makedirs(p, exist_ok=True)

def to_gray(bgr: np.ndarray) -> np.ndarray:
    return cv2.cvtColor(bgr, cv2.COLOR_BGR2GRAY)

def clamp_bbox(x0: int, y0: int, x1: int, y1: int, W: int, H: int) -> Tuple[int, int, int, int
    ]:
```

```
    x0 = max(0, min(x0, W - 2))
    y0 = max(0, min(y0, H - 2))
    x1 = max(x0 + 2, min(x1, W))
    y1 = max(y0 + 2, min(y1, H))
    return x0, y0, x1, y1

def crop(frame: np.ndarray, bbox: Tuple[int, int, int, int]) -> np.ndarray:
    x0, y0, x1, y1 = bbox
    return frame[y0:y1, x0:x1].copy()

def savefig(out_dir: str, name: str) -> None:
    plt.tight_layout()
    plt.savefig(os.path.join(out_dir, name), dpi=250)
    plt.close()

def robust_mad(x: np.ndarray) -> float:
    med = np.median(x)
    mad = np.median(np.abs(x - med))
    return float(1.4826 * mad + 1e-12)


# ----------------------------
# Video read
# ----------------------------

def read_video(video_path: str, max_frames: Optional[int] = None) -> Tuple[List[np.ndarray],
    float]:
    cap = cv2.VideoCapture(video_path)
    if not cap.isOpened():
        raise RuntimeError(f"Could␣not␣open␣{video_path}")

    fps = cap.get(cv2.CAP_PROP_FPS)
    if fps is None or fps <= 0:
        fps = 30.0

    frames: List[np.ndarray] = []
    n = 0
    while True:
        ok, fr = cap.read()
        if not ok:
            break
        frames.append(fr)
        n += 1
        if max_frames is not None and n >= max_frames:
            break
    cap.release()
    if len(frames) < 5:
        raise RuntimeError("Too␣few␣frames.")
    return frames, float(fps)


# ----------------------------
# Motion start detection
# ----------------------------

def motion_frac(prev_g: np.ndarray, g: np.ndarray, thr: int) -> float:
    diff = cv2.absdiff(prev_g, g)
    _, m = cv2.threshold(diff, thr, 255, cv2.THRESH_BINARY)
    return float(np.mean(m > 0))

def detect_motion_start(frames: List[np.ndarray], fps: float) -> int:
    max_search = min(len(frames) - 1, int(MAX_SEARCH_SECONDS * fps))
    prev = to_gray(frames[0])
    streak = 0
    for i in range(1, max_search):
        g = to_gray(frames[i])
        frac = motion_frac(prev, g, MOTION_DIFF_THR)
        if frac > MOTION_FRAC_THR:
            streak += 1
            if streak >= MOTION_CONSEC:
                return max(0, i - MOTION_CONSEC + 1)
        else:
            streak = 0
        prev = g
    return 0


# ----------------------------
# Background + ROI
# ----------------------------

def estimate_background(frames: List[np.ndarray], end_idx: int) -> np.ndarray:
    end_idx = max(1, min(end_idx, len(frames)))
    idxs = np.linspace(0, end_idx - 1, num=min(BG_SAMPLES, end_idx), dtype=int)
    stack = [to_gray(frames[i]).astype(np.float32) for i in idxs]
    bg = np.median(np.stack(stack, axis=0), axis=0).astype(np.uint8)
    return bg

def morph_cleanup(mask: np.ndarray) -> np.ndarray:
    m = mask.copy()
    if TOP_CUTOFF_PX > 0:
        m[:min(TOP_CUTOFF_PX, m.shape[0]), :] = 0
    m = cv2.medianBlur(m, 5)
    k1 = np.ones((OPEN_K, OPEN_K), np.uint8)
    k2 = np.ones((CLOSE_K, CLOSE_K), np.uint8)
    m = cv2.morphologyEx(m, cv2.MORPH_OPEN, k1)
    m = cv2.morphologyEx(m, cv2.MORPH_CLOSE, k2)
    return m

def bbox_from_motion(frames: List[np.ndarray], bg: np.ndarray, start_idx: int) -> Tuple[int,
    int, int, int]:
    H, W = bg.shape[:2]
    xs: List[int] = []
    ys: List[int] = []
    end = min(len(frames), start_idx + ROI_LOOK_FRAMES)
    for i in range(start_idx, end):
        g = to_gray(frames[i])
        diff = cv2.absdiff(g, bg)
        _, m = cv2.threshold(diff, ROI_BG_THR, 255, cv2.THRESH_BINARY)
        m = morph_cleanup(m)
        coords = np.column_stack(np.where(m > 0))
        if coords.size == 0:
            continue
        ys.extend(coords[:, 0].tolist())
        xs.extend(coords[:, 1].tolist())

    if not xs:
        # fallback central strip
        x0, x1 = int(0.35 * W), int(0.65 * W)
        y0, y1 = int(0.05 * H), int(0.95 * H)
        return (x0, y0, x1, y1)

    x0, x1 = min(xs) - ROI_MARGIN, max(xs) + ROI_MARGIN
    y0, y1 = min(ys) - ROI_MARGIN, max(ys) + ROI_MARGIN
    return clamp_bbox(x0, y0, x1, y1, W, H)


# ----------------------------
# Segmentation methods
# ----------------------------

def mask_adaptive(gray: np.ndarray) -> np.ndarray:
    g = cv2.GaussianBlur(gray, GAUSS_BLUR, 0)
    m = cv2.adaptiveThreshold(
        g, 255,
        cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
        cv2.THRESH_BINARY_INV,
        ADAPTIVE_BLOCK, ADAPTIVE_C
    )
    return m

def mask_bg_subtract(gray: np.ndarray, bg_gray: np.ndarray) -> np.ndarray:
    diff = cv2.absdiff(gray, bg_gray)
    diff = cv2.GaussianBlur(diff, (5, 5), 0)
    _, m = cv2.threshold(diff, BG_DIFF_THR, 255, cv2.THRESH_BINARY)
    return m

def mask_combo(gray: np.ndarray, bg_gray: np.ndarray) -> np.ndarray:
    m1 = mask_bg_subtract(gray, bg_gray)
    m2 = mask_adaptive(gray)
    return cv2.bitwise_and(m1, m2)


# ----------------------------
# Contour selection + tracking
# ----------------------------

def circularity(cnt: np.ndarray) -> float:
    area = cv2.contourArea(cnt)
    per = cv2.arcLength(cnt, True)
    if per <= 1e-9:
        return 0.0
    return float(4.0 * math.pi * area / (per * per))

def select_contour(contours: List[np.ndarray],
                   prev_center: Optional[Tuple[float, float]]) -> Optional[np.ndarray]:
    cand = []
    for c in contours:
        a = cv2.contourArea(c)
        if a < MIN_AREA_PX:
            continue
        circ = circularity(c)
        if circ < MIN_CIRC:
            continue
        (cx, cy), r = cv2.minEnclosingCircle(c)
        cand.append((c, a, circ, float(cx), float(cy), float(r)))
    if not cand:
        return None

    if prev_center is None:
        # prefer round+large
        cand.sort(key=lambda z: z[1] * z[2], reverse=True)
        return cand[0][0]

    px, py = prev_center
    # continuity wins
    cand.sort(key=lambda z: ( (z[3]-px)**2 + (z[4]-py)**2 ) - 0.05 * z[2])
    return cand[0][0]

def track(frames_roi: List[np.ndarray],
          fps: float,
          method: str,
          bg_roi_gray: Optional[np.ndarray],
          out_diag: str) -> TrackResult:

    prev_center: Optional[Tuple[float, float]] = None
    misses = 0

    t_list, x_list, y_list, v_list = [], [], [], []
    best_dbg = {"score": -1.0, "raw": None, "mask": None, "overlay": None}

    for i, fr in enumerate(frames_roi):
        t = i / fps
        g = to_gray(fr)

        if method == "adaptive":
            m = mask_adaptive(g)
        elif method == "bg_subtract":
            m = mask_bg_subtract(g, bg_roi_gray) # type: ignore[arg-type]
        elif method == "combo":
            m = mask_combo(g, bg_roi_gray) # type: ignore[arg-type]
        else:
            raise ValueError(method)

        m = morph_cleanup(m)
        contours, _ = cv2.findContours(m, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
        cnt = select_contour(contours, prev_center)

        t_list.append(t)
        if cnt is None:
            x_list.append(np.nan); y_list.append(np.nan); v_list.append(False)
            prev_center = None
            misses += 1
            if misses >= MAX_MISSES:
                break
            continue

        (cx, cy), r = cv2.minEnclosingCircle(cnt)
        cx = float(cx); cy = float(cy); r = float(r)

        if prev_center is not None:
            dx = cx - prev_center[0]
            dy = cy - prev_center[1]
            if dx*dx + dy*dy > MAX_JUMP_PX * MAX_JUMP_PX:
                # reject jump
                x_list.append(np.nan); y_list.append(np.nan); v_list.append(False)
                prev_center = None
                misses += 1
                if misses >= MAX_MISSES:
                    break
                continue

        x_list.append(cx); y_list.append(cy); v_list.append(True)
        prev_center = (cx, cy)
        misses = 0

        # debug: store best "clean" detection (area*circularity)
        area = cv2.contourArea(cnt)
        circ = circularity(cnt)
        score = float(area * circ)
        if score > best_dbg["score"]:
            overlay = cv2.cvtColor(g, cv2.COLOR_GRAY2BGR)
            cv2.drawContours(overlay, [cnt], -1, (0, 255, 0), 2)
            cv2.circle(overlay, (int(cx), int(cy)), 3, (0, 0, 255), -1)
            best_dbg.update(score=score, raw=fr.copy(), mask=m.copy(), overlay=overlay.copy())

    # write best debug images
    if best_dbg["raw"] is not None:
        cv2.imwrite(os.path.join(out_diag, f"processing_{method}_raw.png"), best_dbg["raw"])
        cv2.imwrite(os.path.join(out_diag, f"processing_{method}_mask.png"), best_dbg["mask"])
        cv2.imwrite(os.path.join(out_diag, f"processing_{method}_overlay.png"), best_dbg["
            overlay"])

    t = np.array(t_list, float)
    x = np.array(x_list, float)
    y = np.array(y_list, float)
    valid = np.array(v_list, bool)

    # roi placeholder in TrackResult (filled by caller)
    return TrackResult(method=method, fps=fps, roi=(0, 0, 0, 0), t=t, x_px=x, y_px=y, valid=
        valid)


# ----------------------------
# Physics interval + impact detection (Option B)
# ----------------------------

def longest_monotone_segment(t: np.ndarray, y: np.ndarray,
                             min_points: int,
                             tol_px: float) -> Tuple[np.ndarray, np.ndarray]:
    ok = np.isfinite(y)
    if ok.sum() < min_points:
        return np.array([]), np.array([])

    idx = np.where(ok)[0]
    best = (idx[0], idx[0])
    start = idx[0]
    prev = idx[0]

    for k in idx[1:]:
        # if contiguous and monotone-ish
        if (k == prev + 1) and (y[k] + tol_px >= y[prev]):
            prev = k
        else:
            if prev - start > best[1] - best[0]:
                best = (start, prev)
            start = k
            prev = k

    if prev - start > best[1] - best[0]:
        best = (start, prev)

    a, b = best
    seg_t = t[a:b+1]
    seg_y = y[a:b+1]
    if len(seg_y) < min_points:
        return np.array([]), np.array([])
    return seg_t, seg_y
```

```
def safe_spline(t: np.ndarray, y: np.ndarray, s0: float, max_tries: int = 8) ->
    UnivariateSpline:
    """
␣␣␣␣Build␣a␣smoothing␣spline␣robustly␣by␣increasing␣s␣if␣needed.
␣␣␣␣Prevents␣'maxit␣reached:␣s␣too␣small'␣warnings␣from␣breaking␣logic.
␣␣␣␣"""
    # Ensure strictly increasing t (UnivariateSpline can behave poorly otherwise)
    order = np.argsort(t)
    t = np.asarray(t[order], float)
    y = np.asarray(y[order], float)

    # Remove duplicate/near-duplicate t values by small jitter (rare but helps)
    dt = np.diff(t)
    if np.any(dt <= 0):
        # Make t strictly increasing
        eps = 1e-9
        for i in range(1, len(t)):
            if t[i] <= t[i-1]:
                t[i] = t[i-1] + eps

    s = float(s0)
    last_err = None
    for _ in range(max_tries):
        try:
            return UnivariateSpline(t, y, s=s)
        except Exception as e:
            last_err = e
            s *= 5.0
    # If it still fails, raise the last error
    raise RuntimeError(f"Could␣not␣build␣spline␣even␣after␣increasing␣s.␣Last␣error:␣{last_err
        }")

def detect_impact_index(t: np.ndarray, y_px: np.ndarray, roi_h_px: int) -> Optional[int]:
    """
␣␣␣␣Robust␣impact␣detection:
␣␣␣␣-␣impact␣expected␣near␣bottom␣of␣ROI
␣␣␣␣-␣detect␣either␣(i)␣bounce,␣(ii)␣stop/plateau,␣(iii)␣tracking␣loss␣near␣bottom
␣␣␣␣"""
    ok = np.isfinite(y_px)
    if ok.sum() < 8:
        return None

    idx_all = np.where(ok)[0]
    tt = t[ok]
    yy = y_px[ok]

    # Smooth in pixels (auto-adjust s)
    spl = safe_spline(tt, yy, s0=1e-2)
    ys = spl(tt)
    vs = spl.derivative(1)(tt)

    # Define "near bottom"
    bottom_band = 0.90 * float(roi_h_px) # last 10% of ROI
    near_bottom = ys >= bottom_band

    # (1) Bounce: first sign change + to -
    for i in range(2, len(vs)):
        if vs[i-1] > 0 and vs[i] < 0 and near_bottom[i]:
            return int(idx_all[i])

    # (2) Plateau/stop near bottom: velocity small for consecutive frames
    v_abs = np.abs(vs)
    v_med = np.median(v_abs)
    v_mad = robust_mad(v_abs)
    v_small = max(1.0, v_med - 0.5 * v_mad) # conservative small threshold (px/s)

    consec = 0
    for i in range(len(vs)):
        if near_bottom[i] and v_abs[i] < v_small:
            consec += 1
            if consec >= 3:
                return int(idx_all[i])
        else:
            consec = 0

    # (3) Tracking loss near bottom:
    # If the last valid point is near bottom, treat it as impact.
    last_valid = idx_all[-1]
    if np.isfinite(y_px[last_valid]) and y_px[last_valid] >= bottom_band:
        return int(last_valid)

    # Otherwise: no reliable impact
    return None


def estimate_meters_per_pixel_from_drop(y_px: np.ndarray,
                                        t: np.ndarray,
                                        drop_height_m: float,
                                        roi_h_px: int,
                                        start_skip_frames: int = 0) -> Optional[float]:
    """
␣␣␣␣Compute␣meters_per_pixel␣using:
␣␣␣␣␣␣␣␣y0_px␣=␣median␣of␣early␣valid␣y␣after␣start_skip_frames
␣␣␣␣␣␣␣␣yimp_px␣=␣detected␣impact␣y
␣␣␣␣␣␣␣␣m_per_px␣=␣drop_height_m␣/␣(yimp_px␣-␣y0_px)
␣␣␣␣"""
    ok = np.isfinite(y_px)
    if ok.sum() < 10:
        return None

    idx = np.where(ok)[0]
    idx = idx[idx >= start_skip_frames]
    if len(idx) < 10:
        return None

    # start y: robust median of first few points after skip
    first = idx[:5]
    y0 = float(np.median(y_px[first]))

    # roi_h_px must be passed in by caller; change function signature accordingly
    imp_i = detect_impact_index(t, y_px, roi_h_px)
    if imp_i is None or not np.isfinite(y_px[imp_i]):
        return None
    yimp = float(y_px[imp_i])

    fall_px = yimp - y0
    # Require at least 25% of ROI height of vertical travel
    if fall_px <= 0.25 * roi_h_px:
        return None

    return float(drop_height_m / fall_px)


# ----------------------------
# Method scoring
# ----------------------------

def roughness_metric(y: np.ndarray) -> float:
    if len(y) < 4:
        return float("inf")
    d2 = y[2:] - 2*y[1:-1] + y[:-2]
    return float(np.sqrt(np.mean(d2*d2)))

def quad_fit_rmse(t: np.ndarray, y: np.ndarray) -> float:
    if len(y) < 5:
        return float("inf")
    A = np.column_stack([t*t, t, np.ones_like(t)])
    c, *_ = np.linalg.lstsq(A, y, rcond=None)
    yhat = A @ c
    return float(np.sqrt(np.mean((y - yhat)**2)))


# ----------------------------
# Physics models + fitting
# ----------------------------

def y_no_drag(p: np.ndarray, t: np.ndarray) -> np.ndarray:
    y0, v0, g = p
    return y0 + v0*t + 0.5*g*t*t

def y_linear(p: np.ndarray, t: np.ndarray) -> np.ndarray:
    y0, v0, g, c = p
    c = max(float(c), 1e-9)
    return y0 + (g/c)*t + (v0 - g/c)*(1.0 - np.exp(-c*t))/c

def y_quadratic(p: np.ndarray, t: np.ndarray) -> np.ndarray:
    y0, v0, g, c = p
    c = max(float(c), 1e-9)
    g = max(float(g), 1e-9)
    B = math.sqrt(g*c)
    vt = math.sqrt(g/c)
    z0 = np.clip(v0/vt, -0.999, 0.999)
    A = np.arctanh(z0)
    return y0 + (vt/B) * (np.log(np.cosh(A + B*t)) - np.log(np.cosh(A)))

def fit_model(kind: str, t: np.ndarray, y: np.ndarray) -> Dict[str, object]:
    if kind == "no_drag":
        p0 = np.array([y[0], 0.0, G_INIT], float)
        lo = np.array([-np.inf, -np.inf, 0.0])
        hi = np.array([ np.inf, np.inf, 100.0])
        f = lambda p: y_no_drag(p, t) - y
        k = 3
    elif kind == "linear":
        p0 = np.array([y[0], 0.0, G_INIT, 0.5], float)
        lo = np.array([-np.inf, -np.inf, 0.0, 1e-9])
        hi = np.array([ np.inf, np.inf, 100.0, 50.0])
        f = lambda p: y_linear(p, t) - y
        k = 4
    elif kind == "quadratic":
        p0 = np.array([y[0], 0.0, G_INIT, 0.2], float)
        lo = np.array([-np.inf, -np.inf, 0.0, 1e-9])
        hi = np.array([ np.inf, np.inf, 100.0, 50.0])
        f = lambda p: y_quadratic(p, t) - y
        k = 4
    else:
        raise ValueError(kind)
    res = least_squares(f, p0, bounds=(lo, hi), max_nfev=20000)
    p = res.x
    yhat = y + res.fun
    resid = y - yhat
    rss = float(np.sum(resid*resid))
    rm = float(np.sqrt(np.mean(resid*resid)))
    aic = float(len(y) * np.log(rss/max(1, len(y))) + 2*k)
    return {"params": p, "yhat": yhat, "rmse": rm, "aic": aic}

def fit_all_models(t: np.ndarray, y: np.ndarray) -> Dict[str, Dict[str, object]]:
    out = {}
    for k in ("no_drag", "linear", "quadratic"):
        try:
            out[k] = fit_model(k, t, y)
        except Exception:
            pass
    return out


# ----------------------------
# Plotting
# ----------------------------

def plot_track(out_plots: str, tag: str, t: np.ndarray, y: np.ndarray, title: str) -> None:
    plt.figure(figsize=(7, 4))
    plt.plot(t, y, "o-")
    plt.xlabel("t␣(s)")
    plt.ylabel("y␣(m)" if "m" in title else "y␣(px)")
    plt.title(title)
    plt.grid(True, alpha=0.3)
    savefig(out_plots, tag)

def plot_fits(out_plots: str, tag: str, t: np.ndarray, y: np.ndarray, fits: Dict[str, Dict[str
    , object]], title: str) -> None:
    plt.figure(figsize=(7, 4))
    plt.plot(t, y, "o", label="data")
    for k, d in fits.items():
        plt.plot(t, d["yhat"], "-", label=f"{k}␣(RMSE={d['rmse']:.3g},␣AIC={d['aic']:.2f})")
    plt.xlabel("t␣(s)")
    plt.ylabel("y␣(m)")
    plt.title(title)
    plt.grid(True, alpha=0.3)
    plt.legend()
    savefig(out_plots, tag)

def plot_method_metrics(out_plots: str, metrics: Dict[str, dict]) -> None:
    names = [m for m in metrics if metrics[m].get("ok", False)]
    if not names:
        return
    rmse = [metrics[m]["fit_rmse_px"] for m in names]
    cov = [metrics[m]["coverage"] for m in names]
    rough = [metrics[m]["roughness_px"] for m in names]
    score = [metrics[m]["score"] for m in names]

    def bar(vals, title, ylabel, fname):
        plt.figure(figsize=(7, 4))
        plt.bar(names, vals)
        plt.title(title)
        plt.ylabel(ylabel)
        plt.grid(True, axis="y", alpha=0.3)
        savefig(out_plots, fname)
    bar(rmse, "Method␣comparison:␣quadratic␣fit␣RMSE␣(px)", "RMSE␣(px)", "metric_rmse_px.png")
    bar(rough, "Method␣comparison:␣roughness␣(px)", r"RMS($\Delta^{2}$y)␣(px)", "
        metric_roughness_px.png")
    bar(cov, "Method␣comparison:␣coverage", "coverage", "metric_coverage.png")
    bar(score, "Method␣comparison:␣total␣score␣(lower␣better)", "score", "metric_score.png")

def plot_v_a(out_plots: str, t: np.ndarray, y: np.ndarray, v: np.ndarray, a: np.ndarray, tag:
    str) -> None:
    # v(t)
    plt.figure(figsize=(7, 4))
    plt.plot(t, v, "o-")
    plt.xlabel("t␣(s)")
    plt.ylabel("v␣(m/s)")
    plt.title(f"v(t)␣[{tag}]")
    plt.grid(True, alpha=0.3)
    savefig(out_plots, f"v_t_{tag}.png")

    # a(t)
    plt.figure(figsize=(7, 4))
    plt.plot(t, a, "o-")
    plt.axhline(9.81, linestyle=":", linewidth=1, label="g=9.81")
    plt.xlabel("t␣(s)")
    plt.ylabel("a␣(m/s$^2$)")
    plt.title(f"a(t)␣[{tag}]")
    plt.grid(True, alpha=0.3)
    plt.legend()
    savefig(out_plots, f"a_t_{tag}.png")

    # v(y)
    plt.figure(figsize=(7, 4))
    plt.plot(y, v, "o-")
    plt.xlabel("y␣(m)")
    plt.ylabel("v␣(m/s)")
    plt.title(f"v(y)␣[{tag}]")
    plt.grid(True, alpha=0.3)
    savefig(out_plots, f"v_y_{tag}.png")

    # a(y)
    plt.figure(figsize=(7, 4))
    plt.plot(y, a, "o-")
    plt.axhline(9.81, linestyle=":", linewidth=1, label="g=9.81")
    plt.xlabel("y␣(m)")
    plt.ylabel("a␣(m/s$^2$)")
    plt.title(f"a(y)␣[{tag}]")
    plt.grid(True, alpha=0.3)
    plt.legend()
    savefig(out_plots, f"a_y_{tag}.png")


# -----------------------------
# Main
# -----------------------------

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--video", required=True)
    ap.add_argument("--out", default="out")
    ap.add_argument("--drop_height_m", type=float, default=DEFAULT_DROP_HEIGHT_M)
    ap.add_argument("--skip_after_motion", type=int, default=10)
    ap.add_argument("--max_frames", type=int, default=None)
    ap.add_argument("--force_meters_per_pixel", type=float, default=None,
                  help="If␣set,␣skip␣auto-calibration␣and␣use␣this␣value.")
    args = ap.parse_args()

    out_dir = args.out
    out_diag = os.path.join(out_dir, "diagnostics")
    out_plots = os.path.join(out_dir, "plots")
    ensure_dir(out_dir); ensure_dir(out_diag); ensure_dir(out_plots)

    frames, fps = read_video(args.video, max_frames=args.max_frames)
    H, W = frames[0].shape[:2]

    motion_idx = detect_motion_start(frames, fps)
    start_idx = min(len(frames)-1, motion_idx + args.skip_after_motion)

    bg = estimate_background(frames, motion_idx)
    roi = bbox_from_motion(frames, bg, start_idx)
    x0, y0, x1, y1 = roi

    # crop analysis frames + background ROI
    frames_roi = [crop(fr, roi) for fr in frames[start_idx:]]
    bg_roi_gray = crop(bg, roi)

    methods = ["combo", "bg_subtract", "adaptive"]
    tracks: Dict[str, TrackResult] = {}

    # Track per method (pixel domain)
    for m in methods:
        tr = track(frames_roi, fps, m, bg_roi_gray if m != "adaptive" else None, out_diag)
        tr.roi = roi
        tracks[m] = tr

    # Score each method (pixel domain, using monotone segment)
    metrics: Dict[str, dict] = {}
    for m, tr in tracks.items():
        t = tr.t
        y = tr.y_px.copy()
        y[~tr.valid] = np.nan

        seg_t, seg_y = longest_monotone_segment(t, y, min_points=MIN_POINTS_PHYS, tol_px=
            MONO_TOL_PX)
        if len(seg_y) < MIN_POINTS_PHYS:
            metrics[m] = {"ok": False, "reason": "not␣enough␣points␣in␣monotone␣segment"}
            continue

        y0 = float(seg_y[0])
        y_rel = seg_y - y0 # pixels downward
        fit_rm = quad_fit_rmse(seg_t, y_rel)
        rough = roughness_metric(y_rel)
        cov = tr.coverage
        score = float(fit_rm + LAMBDA_COV*(1.0 - cov) + MU_ROUGH*rough)

        metrics[m] = {
            "ok": True,
            "coverage": cov,
            "fit_rmse_px": float(fit_rm),
            "roughness_px": float(rough),
            "score": score,
            "n_points_total": int(np.sum(tr.valid)),
            "n_points_phys": int(len(seg_y)),
        }

        # Save baseline pixel y(t)
```

```
        plot_track(out_plots, f"baseline_y_px_{m}.png", seg_t, y_rel, f"y(t)␣in␣pixels␣(method
            ={m})")

    # Choose best method
    ok_methods = [m for m in methods if metrics.get(m, {}).get("ok", False)]
    if not ok_methods:
        raise RuntimeError("No␣method␣produced␣a␣usable␣monotone␣physics␣segment.␣Check␣masks/
            ROI/top_cutoff.")

    best_method = min(ok_methods, key=lambda mm: metrics[mm]["score"])
    plot_method_metrics(out_plots, metrics)

    # Best track series for calibration + physics
    best_tr = tracks[best_method]
    t_all = best_tr.t
    y_all = best_tr.y_px.copy()
    y_all[~best_tr.valid] = np.nan

    roi_h_px = frames_roi[0].shape[0]

    # Choose physics segment again for best method
    t_seg, y_seg = longest_monotone_segment(t_all, y_all, min_points=MIN_POINTS_PHYS, tol_px=
        MONO_TOL_PX)
    if len(y_seg) < MIN_POINTS_PHYS:
        raise RuntimeError("Best␣method␣still␣has␣too␣short␣physics␣segment␣(unexpected).")

    # Auto-calibration (Option B)
    if args.force_meters_per_pixel is not None:
        m_per_px = float(args.force_meters_per_pixel)
        cal_source = "forced"
        imp_idx = None
    else:
        m_per_px = estimate_meters_per_pixel_from_drop(
            y_px=y_all, t=t_all, drop_height_m=args.drop_height_m,
            roi_h_px=roi_h_px,
            start_skip_frames=0
        )

        cal_source = "auto_drop"
        imp_idx = detect_impact_index(t_all, y_all, roi_h_px)

    if m_per_px is None:
        # fallback: run in pixel-units but warn
        m_per_px = 1.0
        cal_source = "fallback_px_units"

    # Convert physics segment to meters (downward positive)
    y0_px = float(y_seg[0])
    y_m = (y_seg - y0_px) * m_per_px

    # Spline-smooth y(t) in meters for fitting stability
    spl = UnivariateSpline(t_seg, y_m, s=SPLINE_S)
    y_s = spl(t_seg)

    v_s = spl.derivative(1)(t_seg)
    a_s = spl.derivative(2)(t_seg)

    plot_v_a(out_plots, t_seg, y_s, v_s, a_s, tag=f"{best_method}_spline")

    # Fit models
    fits = fit_all_models(t_seg, y_s)
    if not fits:
        raise RuntimeError("All␣physics␣fits␣failed.")

    best_model = min(fits.keys(), key=lambda k: fits[k]["aic"])

    # Plots
    plot_track(out_plots, "baseline_y_m_best.png", t_seg, y_m,
              f"y(t)␣in␣meters␣(best␣method={best_method},␣meters_per_pixel={m_per_px:.6g})")
    plot_fits(out_plots, "fits_y_m.png", t_seg, y_s, fits,
             f"Model␣fits␣(best␣method={best_method},␣best␣model␣by␣AIC={best_model})")

    # Save summary
    summary = {
        "video": args.video,
        "fps": fps,
        "frame_count": len(frames),
        "motion_start_frame": motion_idx,
        "analysis_start_frame": start_idx,
        "roi": {"x0": x0, "y0": y0, "x1": x1, "y1": y1},
        "best_method": best_method,
        "method_metrics": metrics,
        "drop_height_m": args.drop_height_m,
        "meters_per_pixel": m_per_px,
        "calibration_source": cal_source,
        "impact_index": int(imp_idx) if imp_idx is not None else None,
        "best_model_by_aic": best_model,
        "fits": {k: {"params": [float(v) for v in fits[k]["params"]],
                    "rmse": float(fits[k]["rmse"]),
                    "aic": float(fits[k]["aic"])} for k in fits},
        "NOTE": "Auto-calibration␣assumes␣y_start␣and␣y_impact␣correspond␣to␣the␣known␣drop␣
            height.␣Perspective␣can␣bias␣scale."
    }
    with open(os.path.join(out_dir, "run_summary.json"), "w", encoding="utf-8") as f:
        json.dump(summary, f, indent=2)

    print(f"[OK]␣Outputs␣in:␣{out_dir}")
    print(f"[OK]␣Best␣method:␣{best_method}")
    print(f"[OK]␣Calibration␣source:␣{cal_source}")
    print(f"[OK]␣meters_per_pixel:␣{m_per_px:.6g}")
    if cal_source == "fallback_px_units":
        print("[WARN]␣Could␣not␣detect␣impact␣reliably␣->␣running␣in␣pixel-units.␣Improve␣
            visibility␣of␣floor/impact.")
    else:
        print(f"[OK]␣Using␣drop␣height␣{args.drop_height_m}␣m␣to␣infer␣scale.")
    print(f"[OK]␣Best␣model␣by␣AIC:␣{best_model}")


if __name__ == "__main__":
    main()
```