Object movement prediction through visual perception and learning of physical properties

Liviu Daniel Florescu

BSc Computer Science, Year 3

Supervised by:

Dr. Hongkai Wen

Mr. Christos Makris

Department of Computer Science

University of Warwick

May 2021

Abstract

Psychological research indicates that the abstract cognitive model used by humans for reasoning about physical phenomena can be interpreted as a noisy simulation medium. This project aims to implement an equivalent system, using a physics engine and Markov Chain Monte Carlo methods to infer physical properties of objects. A simple physical scenario, the inclined plane, is observed in unlabelled videos from a public dataset, and the velocity of the objects in the scene is extracted through a tracking algorithm. The velocity profile serves as a physically-interpretable comparison space, which the model explores by continuously proposing object parameters and simulating the scene with the given proposals. Experimental results show that the model is able to infer the values of the parameters which directly influence the features of the velocity profile. For comparison, a Convolutional Neural Network with the same purpose was designed, outperforming the probabilistic model and learning the intrinsic parameter equations which determined the observed behaviour, yet requiring dedicated training data and introducing an additional latent space.

Keywords: physical properties, probabilistic inference, tracking, velocity profile, physics engine, simulation

For now we see only a reflection as in a mirror; then we shall see face to face.
Now I know in part; then I shall know fully...
1 Corinthians 13:12

Contents

List of Figures

Chapter 1

Introduction

What is reality?

Even the simple pondering of this question raises numerous dilemmas related to the nature of our surroundings, our perceptive abilities, and the processes through which the environment becomes the abstract entity about which humans are able to reason. It is beyond the purpose of this work to provide an answer, but we would like to preserve this puzzling concept and bring it into the light of technological methods. As soon as we approach this notion, we uncover the need for observation – whether this is embedded in our genes, or it lies in the nature of reality, it shall remain a problem for the philosopher –, expressed in a witness whose senses probe, record and analyse the environment.

From a very wide, yet concrete perspective, technology is the physical, immediate manifestation of human adaptation. Any purposeful tool simplifies our activity and reduces the effort required to complete a given task. Be it an ancient wheel or a computer system, the essence of these tools is similar. However, as time went by, the complexity of the activities that tools can perform increased steadily. After mechanised mass production replaced highly repetitive tasks, the next milestone involved a paradigm shift, with regards to how they interact with the environment: from passive to active operation – an evocative example are vacuum cleaners [17]. This uncovers a new category of lifeless, artificial witnesses, whose relation to reality mimics the human perceptual process.

Artificial systems interacting with the world

The key aspect is that science has the overwhelming potential to control this relation, by altering its components. Even in a non-alterable environment, a technology-driven witness will be presented with a different representation, as a function of the sensing mechanism and the interpretation method. But this highlights yet another design question: should the interaction model (here, broadly defined as the environment – sensing – intelligence tuple) of an artificial witness actually aim to imitate the human one? More or less surprisingly, most of the significant advancements in technology seem to exploit some property specific

to natural processes. Technology is, metaphorically, bound to replicate nature.

In order to replicate it, however, an in-depth understanding is necessary. Even if the general dilemma of humans' relation to reality dates back to classical antiquity, it was only relatively recent in our history that scientific work targeted the interpretation of human understanding of physical events. Consequently, even less effort has been put into translating this interpretation into a technical model, although its benefits and applications are usually trivial to identify – it is not hard to imagine the computational advantage of a system that reasons about the environment up to a complexity comparable to human reasoning, over a simply automated system that, in the same environment, is limited by its sensorial representation. Humans effortlessly see, feel and know more than reality directly presents to them.

In our work, we would like to explore a paradigm for visual interpretation that aims to approach human intuition and infer physical properties of the objects being observed. More specifically, this paradigm is applied to a basic physical scenario (the inclined plane) and an associated estimation problem is formulated, to assess its capabilities. For comparison, we design a Neural Network-based model for the same task, highlighting the differences in the natures of this two distinct approaches. We obtain encouraging results which demonstrate that the models correctly identify and evaluate the parameters which determine scene dynamics, even without explicitly modelling the underlying equations - indicating that a more complex architecture for analytical modelling is not necessary, unless a more involved scenario or a more in-depth analysis are targeted.

1.1 Motivation

We will now bring forward the key elements that constituted the motivation behind deciding to invest effort into this subject. There are two main motivation vectors underlying the project, each targeting a different aspect of the matter.

Can this goal be achieved?

The first one focuses on the final goal of the project, also considering the bigger picture with which it aligns. Given an artificial system and its environment, it is often the case in practice that visual data provides the highest amount of information at the cheapest cost. Therefore, harvesting this information becomes key to environment understanding and consequently, to most interactions. Additionally, it has become trivial that newer systems, aiming for breakthroughs in the technology-as-a-tool field, increasingly employ autonomous processes. This tendency is natural, considering that the tasks to be performed were, at their origin, only suitable to humans - with the most relevant example being self-driven vehicles. Such progress has a wide range of applications, from logistics

(transportation, inventory management, warehousing) to hospitality and leisure [7].

The specific task of physical environment perception is also motivated by practicality. In the fortunate circumstance that actual physical properties are inferable from visual data alone, with satisfactory accuracy and efficiency, engineers will find themselves in the position to replace complicated sensor systems with a combination of cameras and processing units, as it is already the case with distance estimation [18] (figure 1.1). Universal solutions are, certainly, not a feasible goal for the very close future, but we are confident that by designing and developing for local problems, such systems have the potential to bring considerable improvements.

(a) Real-world input data (b) Output data - predicted depth

Figure 1.1: Depth estimation using a Convolutional Neural Network
No ground truth data is necessary for training. Results by Godard et al. [18].

An unconventional approach

The second motivation vector is closely related to the intrinsic method that we are going to use - in other words, how the goal mentioned earlier will be reached. One vital aspect is that, in the real world, repetitive patterns are common. We very often have the opportunity to observe an event occurring continuously: cars passing by on the street, birds flying, waves hitting the shore etc. In a controlled artificial environment, repetition is also trivial to generate: robots can perform an action multiple times. This highlights a feature that our investigation will exploit, by extracting information from a source that, from this perspective, imitates daily processes. Furthermore, we are faced with the challenging problem of exploring a basic physical system which stands as a prototype for more complex interactions. Nonetheless, even in this simplified model, object properties have a strong influence on the interactions, and it is this link between behaviour and intrinsic characteristics that we are interested in.

Another strongly motivating point is the rather unique approach to learning. We would like to note that, unsurprisingly, computers are prone to use human-imitating techniques

for tasks that involve cognitive processes. This idea is empirically supported by the results of Deep Learning methods, which aim to artificially mimic the neural mechanism. Instead of focusing on the neural model, we explore the capabilities of a model that operates on data in a high-level fashion that is closer to human judgement. Wandering aside from the widespread, popular approaches, this exploration has the potential to uncover interesting findings.

Chapter 2

Background

In this section we will describe significant progress which has been made in the areas that closely influence our project. These areas are related either through the applicative side of the technologies being involved, or directly through the methods that were used. A key resource is the Galileo system [47], published in 2015, which stands as a stepping stone for our design. This paper proposed a shift from the common Deep Learning paradigm in the field, by introducing a less explored connection to psychology. From a psychological point of view, we understand very little about how humans reason about the physical processes in their surroundings. This is an active research area and has a natural link to a question in modern day autonomous processes: how should machines reason about the world? More specifically – should we force machines to follow human-like thinking, or is there a more viable alternative out there? Extensive work has also been put into investigating the relationship between intuitive and theoretical Newtonian mechanics, as understanding this connection would bring us closer to an answer to the questions above.

Understanding and modelling human reasoning

In his 1943 work *The Nature of Explanation* [16], Kenneth Craik supported the idea that humans use 'runnable' mental simulations for outcome prediction. This is a very natural process: when we look at a scene, e.g. a set of stacked books, we immediately identify the independent elements and unconsciously assess their stability, consequently inferring some potential outcomes of the scene – will they fall, when nobody touched them? Will they fall, if slightly pushed?

Peter W. Battaglia et al. [4] specifically targeted questions of this type. They also provided an in-depth review of how our understanding of human physical scene interpretation has evolved through the years – in the early stages [24], experiments faced the issue of unexplainable error patterns in human intuition, when presented simple one-body systems. This put forward the potential explanation that intuitive physics are ultimately incompatible with the principles of theoretical Newtonian mechanics. But progress did

not halt there. An alternative explanation which emerged later on is that the erroneous results are caused by the noisiness of observations. Apart from external noise (e.g. differences in perception), humans are subject to intrinsic noise – no two neural processes will ever be truly identical, even more so in distinct subjects.

"*How else might people's physical scene understanding work, if not through model-based simulation?*"

Based on this question, [4] also introduces a computational framework which aims to mimic the intuitive physical inferences that humans make, focusing on applied scenarios typical for everyday life (where a large number of objects are present, observations are incomplete and interactions are complex). Detailed representation and accuracy are exchanged for speed, to enhance the human-like behaviour. Extensive experiments which compare the virtual physics simulator with human performance were run, and results were encouraging. The model performed better when predicting people's judgements, compared to a feature-based equivalent (representing the alternative theory that judgements are based on learned geometric characteristics of the scene, and not actual physical interactions), while specific experiments supported the point that probabilities also play a key role – this relates to the noisiness incurred by intuition.

Adam Sanborn [34] was also intrigued by the multitude of facets in this problem, which he traced back to 1963, when Michotte published a major analysis on *The Perception of Causality* [25], using experiments and human observations of relatively simple object collisions as evidence for his investigation of how intuition and Newtonian physics are related. Sanborn's work aimed to reconcile these apparently contrasting frameworks and concluded that people's inferences can indeed be explained in a manner that is consistent with Newtonian laws, provided that Bayesian inference is involved in order to take into account the noise in people's observations. This highlights two important aspects: discussions about the origins of uncertainty in human perception are vital, and the fact that the visual system is, in itself, victim of uncertainty that can be modelled in a Bayesian way.

Tomer Ullman et al. [43] approached this area from a different direction: how should a program learn about object interactions in new scenes? Guided by the behaviour of human learners, the resulting model made use of both Bayesian learning and heuristic inference, further emphasising these key principles that underpin the human physical intuition.

Moving towards the goal of enabling artificial systems to reason about physical scenarios, a crucial advancement was presented by [3], in the form of two related contributions. First, an abstract model for physical systems. Here, the building blocks are objects and their relationships, organised in a graph, while dynamics are modelled by either object-

centric functions (dynamics of independent objects) or relation-centric functions (dynamics of object interactions). The second contribution is a DL architecture which implements this abstract model and achieved significant performance on tasks of dynamics prediction and estimation of properties.

Extracting physical information using Deep Learning

Up to this point, we have not touched on yet another key issue: is visual data sufficient to extract physical information from a scene? To some degree, it is, and relevant examples exist in e.g. the recent work in depth estimation [18]. However, Roozbeh Mottaghi [28] stated that "*Computer Vision literature does not provide a reliable solution to direct estimation of mass, friction, the angle of an inclined plane, etc. from an image*". His work was aimed at this missing piece of our technological puzzle and targeted the challenging goal of estimating the forces that are applied on an object in a static image. 12 representative "Newtonian scenarios" are defined, to illustrate common dynamical scenes that are of interest. An associated dataset was also built, to depict these scenarios both in a simulated world (note the use of simulation) and in a corresponding natural context. Afterwards, a complex Convolutional Neural Network is used, which takes as input an RGB image (masked to localise the object of interest) and a set of raw, simulated Newtonian scenarios. The purpose of the network is to identify the scenario to which the image belongs, and consequently represent the forces which act on the object (without regard for magnitude, however). This approach yields impressive results, but seems to mainly rely on the abstract features that the convolutions extract from the data, without an actual physical interpretation. Nonetheless, we would like to note that the inclined plane is one of the scenarios this work is considering.

To overcome the blurry aspects involved in a system that relies on DL for prediction of physical attributes or components, [49] proposed an Interpretable Physics Model, in the form of a network whose latent space represents physically-relevant information. The underlying assumption is that humans use this information when reasoning about physical scenes, and thus they should be of interest for a virtual system, as well. Of course, humans cannot accurately estimate meaningful variables such as mass or friction coefficients, but our observations are highly connected to our perceptions of these characteristics. The model is shaped like an Encoder-Decoder network, where the components in the middle of the architecture encode four distinct qualitative measures: mass, speed, friction and other intrinsic information. During training, these constraints are controlled by a loss that enforces objects with the same mass (for example) to lead to similar mass representations in the latent space. The training data is again generated by a simulation engine, and the main application of the NN is to predict future frames in a given scenario. No analysis is made about usage of this method on real-world data, yet it is feasible to believe that,

in a limited context, with a sufficient amount of data and potentially by increasing the complexity of the network, it can achieve similar results outside the virtual environment it was initially built for.

Interaction with the physical world

Another dimension of the problem tackles system interaction with the environment, as this is often the reality in autonomous systems. Interaction is possible and exploiting its benefits is an easily attainable advantage. Broadly replicating principles specific to infant behaviour, [1] explores the case of a robot poking unknown objects (by abstraction, a new environment) and learning the effect of its actions on the placement of those objects. Again, a NN is used, this time in order to learn the mapping between actions and their corresponding effect in the visual state. Although the objects that the robot interacts with are relatively small in size and without representative physical properties (i.e. interacting with them yields no significant force feedback, which would be useful in assessing their physical characteristics), the experimental results show that apart from location, the model also learns some information about object geometry.

A contrasting principle is explored by [29] in a simulated environment that more accurately depicts real-life scenes. If visual data provides enough information to assess physical elements, it should also be enough to infer the effect of certain physical interactions, without effectively performing those interactions. Using the data in the SUN RGB-D dataset [39] and synthesising it into a physics engine, a new dataset, which shows the effect of a force vector being applied on a specific object, is generated. This is then fed into an architecture that, given an image, tries to predict the effects of some external force acting in the scene. The results are encouraging, potentially pointing towards a latent understanding of the scene, although one is not explicitly formulated.

Physical properties from passive observation

We will now briefly point out two works that very closely resemble our strategy, ultimately serving as inspiration and building blocks for our proceedings. The first of these [6] aims to determine the physical parameters by analysing videos of objects in free flight. The system is divided into three components: a preprocessor, which extracts the silhouette of the object in each video frame, by segmenting it against the background; a rigid body simulator – the shape of the objects is known, and thus the trajectory and pose of the object as a function of time are defined by a set of motion equations, parametrised by the characteristics of the object; finally, a gradient descent optimiser aims to minimise the error between the silhouettes observed in the video and the silhouettes simulated mathematically. In practice, experiments uncovered a noisy search space, containing many local minima, while the choice of silhouettes as the optimisation metric, despite its simplic-

ity, compared to object tracking methods, for example, brought to light an unexpected issue: multiple poses of symmetric objects can project to similar silhouettes, confusing the system. One experiment involved predicting the motion of an object by observing only a small portion of the video clip and trying to infer the upcoming poses. The model performed satisfactorily, but it was highlighted that even small errors, inevitable when little input data is available, propagate to larger errors, as the simulation advances and more frames are predicted. In another experiment, the model was also given the mission to infer the direction of the gravity vector and correct it. Overall, this work innovated by its specific focus on object parameters as a hidden characteristic and the idea of bringing together real observed data (video) and a simulation-like medium, albeit an abstract, mathematical one.

The Galileo System

The second inspiration point is the work mentioned at the beginning of this chapter, the Galileo system [47]. It can be seen as a modern reiteration of the methods described above, as it follows a similar structure but employs newer and more effective technologies, while also providing a stronger grounding in the neurological processes that the system targets to imitate – specifically, that humans use an internal simulation model whose noisiness can be expressed in a probabilistic fashion. The associated dataset depicts objects falling down an inclined plane of varying angle, and also involves collisions: a second object is present at the bottom of the ramp. This time, velocity is used as a search space and the problem is formulated from the perspective of Bayesian inference, instead of optimisation. Rather than directly searching for an optimal solution, the search space is portrayed as a distribution over the values of the object properties, and a MCMC technique helps with sampling. The objective function therefore becomes a likelihood computation which guides the Markov chains during exploration of the search space. Additionally, simulation is supported by a dedicated engine, in line with modern advancements, which provides the much-needed balance between parametrisation and generalisation – virtually any real-life scene can be portrayed, and object features are fully adjustable. Once acceptable results are reached, several experiments correlated the performance of the model with results obtained from human observations, in order to test the underlying hypothesis regarding intuitive physics.

Learning more than object properties

Physical object properties, however, are mere parameters which influence object interactions and dynamics. An even more significant analysis can be performed if, instead, focus is placed on identifying the mathematical models which, given these variables, determine the dynamics of a scene.

One class of representative solutions for this problem uses symbolic regression as a means of exploring the space of analytical representations [9] [35]. In such cases, the learned dynamics are not constrained to satisfy any physical principles, but learning makes use of the abundance of available data to identify correlations between observations and the underlying structure.

A more recent solution, described in 2020 by Yu et al. [50], tries to balance between *a priori* structuring of the model and the wide parameter search space, and harvests the flexibility of Neural Networks as function approximators. This is applied to the field of dissipative systems, which follow the Onsager principle, which constraints the possible model structures without effectively reducing the generalisation capabilities of the model.

Nonetheless, we note that the case we are considering is definitely a simpler physical scene, and such methods would introduce a mismatch between this and the method complexity. Still, the ability to identify the analytical model behind physical scenarios constitutes a powerful tool for autonomous systems, as it enables an even deeper understanding of what is being observed, when the task requires a thorough interpretation.

Deep Voxel Flow - Visual outcome prediction

From the applicative side, one problem that our system can address is that of scene outcome prediction. At the visual level, this is usually formulated as video frame synthesis (in our case, extrapolation). The state-of-the-art model for pixel-wise frame extrapolation is Deep Voxel Flow [21], which combines old-school techniques for optical flow with modern DL pixel value estimation. The obtained results are impressive, yet the emphasis is placed on superficial visual information, as there is no explicit focus on the underlying physical information that could be exploited in a controlled environment.

Chapter 3

Project overview

3.1 Context

This section aims to present several aspects that will allow the reader to better grasp the overall approach. Particularly, we will look at the relevance of the scene that the project has in focus, and provide a brief description of the selected dataset.

3.1.1 The inclined plane

A trivial physical scenario, the inclined plane is a component of everyday life whose importance is often underestimated. It appears naturally all over the world, in the form of slopes and hills, and history notes that it is one of the oldest systems [10], if one may call it so, that humans used for mechanical advantage – defined as the measure of force amplification achieved by using a specific device. Despite its popularity and repeated attempts to formulate the physical interpretation behind the phenomenon associated with objects moving along a ramp, it was not until the Renaissance period that it was mathematically solved [8]. Consequently, it was included in the list of six classical simple machines [2], alongside the lever, wheel and axle, pulley, wedge, and screw. From a practical perspective, its main advantage is the transformation of a lifting force (required to move an object upwards) into a pushing/pulling force, that is generally more convenient.

However, in the scene that this project considers, the only external force that acts on the system composed of an object on top of an inclined plane is gravity. In a 2D world, the ramp can be defined in multiple ways, using combinations of its dimensions, but its key characteristic is the angle of inclination. In a 3D world, a depth (width) component must also be specified. The relation between this angle and the friction coefficient between the surface of the plane and that of the object determines the behaviour of the object, when it is placed on the ramp and released. Figure 3.1 provides a graphical representation of the forces involved.

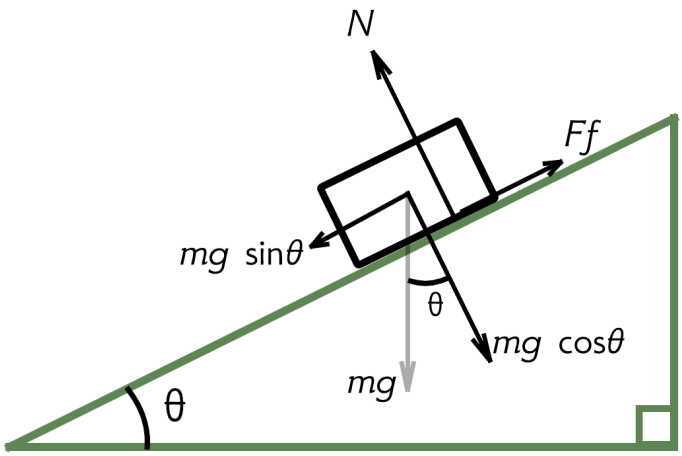


Figure 3.1: Forces acting on a free object, on an inclined plane
m is the mass of the object, g is the gravitational acceleration, θ is the angle of the ramp. N is the normal force and F_f is the friction force between the two surfaces.

The sliding stage

Following the classical method, the gravitational force is decomposed with respect to the plane angle, resulting in two orthogonal vectors – $mg\cos\theta$, perpendicular to the plane surface, which implies a normal force N, and $mg\sin\theta$, parallel to the plane surface, which, when the object is released, will have opposing orientation, with respect to the friction force.

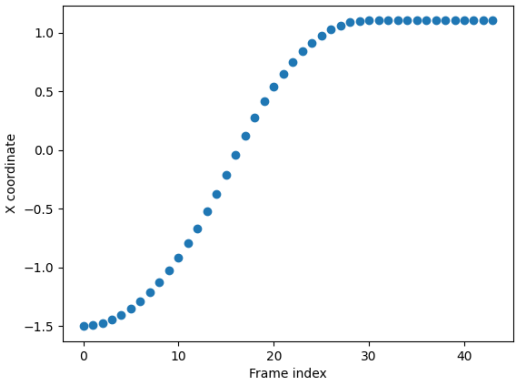
At equilibrium, if air friction is ignored, the following conditions should hold:

$$\begin{aligned} N &= mg\cos\theta \\ mg\sin\theta &= F_f \end{aligned} \tag{3.1}$$

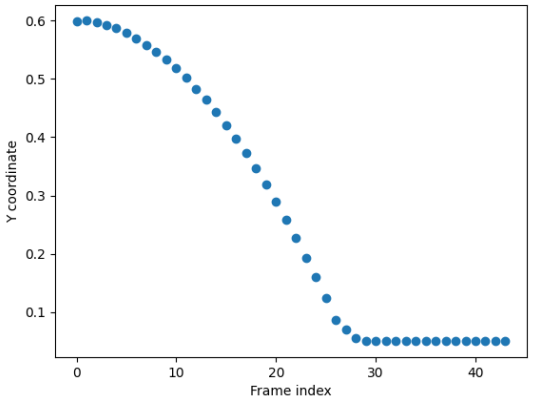
Let us note, however, that the first condition is always true, by the definition of the normal force. In consequence, the only relation left to analyse is the one involving the friction force. This can be rewritten as $mg\sin\theta = \mu N$, again by the definition of F_f, where μ is the friction coefficient between the two surfaces involved. Replacing the value of N, we obtain $mg\sin\theta = \mu mg\cos\theta$, which reduces to $\sin\theta = \mu\cos\theta$. This leads to the conclusion that, when $\mu \geq \tan\theta$, the object will remain in its position and not slide down the ramp. In the world of simple machines, this principle translates to the self-locking property [19]. On the contrary, when $\mu < \tan\theta$, the object will start descending, with an acceleration equal to $a = F/m = (mg\sin\theta – \mu mg\cos\theta)/m = g\sin\theta – \mu g\cos\theta$.

An essential point here is that the sliding condition, and somehow consequently the sliding acceleration, does not depend on the mass of the object. This might appear counterintuitive but can be explained by the human tendency to associate mass to higher friction. In the general case, on a horizontal surface, the force required to move an object indeed depends on its mass, which affects the magnitude of the friction force vector, but this dependency disappears in the isolated system presented above.

As the object slides down the ramp, its trajectory is predetermined by the inclination angle and the starting height. If the bottom of the inclined plane is extended with a horizontal plane of the same friction coefficient, the object will begin to slow down once this joining point is reached. Figure 3.2 shows this behaviour with the help of two plots: x and y coordinates are mapped against time.



(a) Data for the x coordinate. The bottom of the ramp is at $x = 0$, and the object moves towards positive x.



(b) Data for the y coordinate. The bottom of the ramp is at $y = 0$, and the object eventually slides along the horizontal plane.

Figure 3.2: Trajectory of object

Collision analysis

To expand the field of physical phenomena that this scenario offers, it is enhanced with a second object, placed near the bottom of the ramp. This adds a new type of interaction, in the form of the potential collision of the two free objects – the ramp is considered fixed. Depending on the relation between the masses of the two objects, this can result in several behaviours.

By the conservation of total momentum, we have $m_1u_1 + m_2u_2 = m_1v_1 + m_2v_2$, where m_1, m_2 are the masses, u_1, u_2 the initial velocities, and v_1, v_2 the velocities post-collision, of the two objects, respectively. As the second object is static, $u_2 = 0$, the relation becomes $m_1u_1 = m_1v_1 + m_2v_2$ (the initial velocity of the object sliding down from the ramp is distributed to the two objects). Likewise, the conservation of the total kinetic energy indicates that $\frac{m_1u_1^2}{2} + \frac{m_2u_2^2}{2} = \frac{m_1v_1^2}{2} + \frac{m_2v_2^2}{2}$, which can be simplified to $m_1u_1^2 = m_1v_1^2 + m_2v_2^2$.

Solving these two equations, we obtain $v_1 = \frac{m_1-m_2}{m_1+m_2}u_1$ and $v_2 = \frac{2m_1}{m_1+m_2}u_1$. This indicates that, when $m_1 > m_2$, the sliding object will also cause the static object to start moving. When the masses are equal, object 1 will stop and object 2 will "take over" the movement. If $m_2 < m_1$, object 2 prevents object 1 from pursuing its sliding behaviour, potentially causing it to bounce back.

In these equations, friction is disregarded, as we are only considering the instanta-

neous energy modifications of the two objects. The friction between the objects and the horizontal plane will naturally continue to incur negative acceleration, following the collision.

3.1.2 The dataset

In a real-world application of this project, the system would observe the environment and perform the inference actions on visual data obtained from a given setting, which can be represented in a simulated world, to some degree of accuracy. To develop and explore its capabilities, however, a simpler scenario suffices, and this is where the inclined plane comes into action.

The project is built around an impressive public dataset, Physics 101 [46], a joint effort of people from the Massachusetts Institute of Technology and Stanford University, with the aim to provide a solid learning base for any Computer Vision system facing tasks in the area of physical object interactions. The underlying motivation is that humans' physical understanding is developed during early childhood by observing different events, and it is therefore vital to have this type of resource available for computer systems. As noticed in the earlier sections, simple interactions are determined by intrinsic object characteristics – the reverse challenge is to discover these characteristics by observing the interactions as they happen.

The dataset consists of videos depicting five distinct scenarios:

1. Ramp – the situation that our project analyses; the scene is composed of an inclined plane, extended with a horizontal one. Two objects are present – one on top of the inclined plane, and one on the horizontal plane.

2. Spring – a spring is attached to a fixed structure, and an object is hung from the lower end of the spring; this allows the exploration of object mass in an oscillatory setting.

3. Fall – a trivial yet key scenario for understanding physical events, an object is dropped from certain height and falls on different surfaces.

4. Liquid – an object is dropped into a transparent water container; this can prove useful for studies of buoyancy in relation to object density.

5. Multi – an extension of the ramp scenario, with three objects instead of two – an additional object is placed on the horizontal plane.

To increase the number of videos and the variance of the events to be observed, the authors used a wide variety of objects, classified by their materials or type, implying a range of friction coefficients and densities, as such: cardboard, dough, foam, hollow

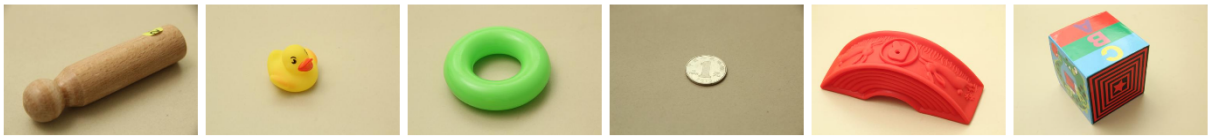


Figure 3.3: A few of the objects present in the dataset
Depicted here are: wooden pole, plastic toy, plastic ring, metal coin, hollow rubber, cardboard box. Note - the scale is not consistent.

rubber, hollow wood, metal coin, metal pole, plastic block, plastic doll, plastic ring, plastic toy, porcelain, rubber, wooden block, wooden pole. Figure 3.3 shows some of these objects, photographed individually. Additionally, each scenario is available in multiple settings: for the spring case, for example, two springs with different stiffness coefficients are used. In the ramp scene, two components can vary: the angle of the inclined plane, and the object that is placed at the bottom of the ramp. This results in four combinations:

1. 10° inclination, a green card box on the
2. 10° inclination, a piece of Styrofoam on the table
3. 20° inclination, a green card box on the table
4. 20° inclination, a piece of Styrofoam on the table

During our experiments, the most used scenario was the third one, as higher inclination usually incurred object movement, while the green card box provided higher contrast with the environment and was thus easier to track.

Each scene setting is available in multiple trials (multiple runs of the same arrangement), to reduce potential bias of corner case situations occurring. If a single result is targeted for each arrangement, an aggregation approach can be used to generate it, after observing trials independently. For each trial, five videos in `.mp4` file format are available, described in table 3.1. These represent exactly the same interaction, from different viewpoints and camera types. Figure 3.4 shows the first and last frame of a `Camera_1.mp4` video for the ramp scenario, which exemplifies the type of input information required for our model. The object on the ramp is held by hand and released as soon as the video recording begins. Video duration is variable, depending on how long the objects move. When no movement occurs following the object being released, the video file will naturally be relatively short.

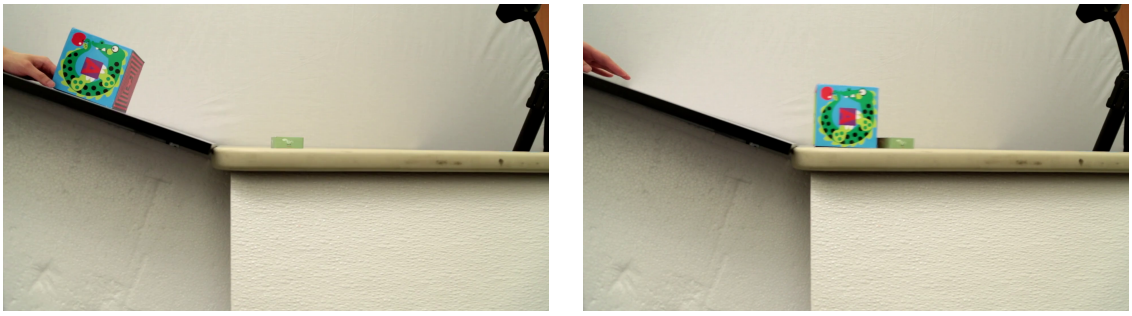
An important note is that these videos, or the objects that appear, are not explicitly labelled in any way. This provides an even more realistic problem, as all the necessary information must be extracted from visual cues. However, we have exploited the folder structure to infer some of the scene/object properties, in order to reduce the visual interpretation burden and focus on the physical properties. This will be described in more detail in the implementation section.

Overall, the dataset includes 3038 trials, out of which 1802 correspond to the ramp

Name	*View*	*Format*	*Camera*	*Resolution (px)*	*Framerate (fps)*
`Camera_1.mp4`	Front	RGB	DSLR	1920x1080	29.97
`Camera_2.mp4`	Side	RGB	DSLR	1920x1080	29.97
`Kinect_FullDepth_1.mp4`	Upper Front	Depth	Kinect V2	512x424	30
`Kinect_RGB_1.mp4`	Upper Front	RGB	Kinect V2	1920x1080	30
`Kinect_RGB-D_1.mp4`	Upper Front	RGB*	Kinect V2	512x424	30

*This video is registered with `Kinect_FullDepth_1.mp4`. Combining these videos would result in an RGB-D file.

Table 3.1: Dataset video file types



(a) The first frame of the video (b) The last frame of the video

Figure 3.4: Sample snapshots from the dataset

scenario (9010 videos). The high amount of visual data available also makes it feasible for a data-hungry model to be employed. An example is the Neural Network developed by [47], which was trained to associate object properties inferred probabilistically to video snapshots, essentially creating a direct mapping from objects' visual aspect to their physical characteristics. Another relevant experiment would consist in training the [21] model on a subset of the data, targeting direct visual predictions of object interaction outcome and implicitly disregarding the physical facet of the problem.

3.2 Requirements

Although this is not a classical software engineering project, pinning down several requirements is an important milestone which will later help with creating a design, testing and evaluating the final system. The overall goal can be expressed in simple words as follows: to design and implement a system that is capable of approximating physical properties of objects seen in a video. Additionally, to maintain a strategy that follows principles considered key to human reasoning, an artificial counterpart of the scene observed in the video must be constructed in a physics simulation engine. Parameter inference will then be conducted using this simulation world and probabilistic sampling of the parameter

value space. Once parameters are estimated, they can be fed back into the simulation engine to generate future states, leading to movement prediction.

Functional and non-functional requirements are detailed in the list below. The border between these two categories is not strict. Some of the functional requirements which incur design constraints can be seen as non-functional, but the current layout emphasises the desire to mimic human principles.

Functional requirements

1. The system must take, as input, a path to a `Camera_1.mp4` video file from the *Physics 101* dataset.
2. The system must output a set of 4 values: m_1, k_1, m_2, k_2 corresponding to the predicted properties of the two objects of interest.
3. The system must use a 3D physics engine.
4. The system must convert the scene in the video to a virtual 3D scene.
5. The system must be able to simulate the scene with varying parameters.
6. The system must take guesses at object properties.
7. The system must determine how close its guesses are to the ground truth (represented by the video)
8. The system must use a probabilistic method to improve its guesses.
9. The system must be able to extrapolate and generate information about future object states, not visible in the video

Non-functional requirements

1. The system should be runnable from a command-line interface

In terms of software and hardware requirements, these are in close relationship with design and implementation decisions, and will become clear as the system unfolds. However, the goal does not necessarily involve creating a distributable system.

Chapter 4

The Bayesian inference model

4.1 Design

This section will define the architecture of the system independent of implementation details, focusing on the main components and their relationships, in order to understand the data flow and how it aims to provide the artificial counterpart of human reasoning.

The initial stage of the design process virtually consisted of the background reading in the area, studying existing solutions and their features, with particular emphasis on the work of the Galileo team [47]. A thorough analysis of their approach was performed, and this proved very useful throughout the entire design phase.

The final design of the system is illustrated in figure 4.1, and it has a specific symmetrical architecture, driven by the fact that its core consists of a comparison step. The top and bottom branches join there, and the result of the comparison is then processed. Ultimately, data follows a tree-like route through the system, but a cyclic behaviour is necessary during execution. This will be covered in depth in the following paragraphs.

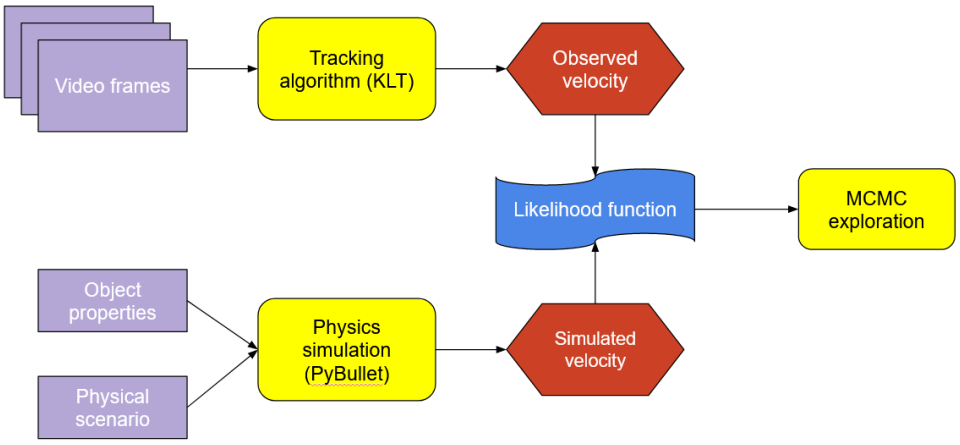


Figure 4.1: System design

The two branches of the system can be perceived as distinct, yet equivalent processing pipelines for two representations of the world: the visual representation and the simulated

one, respectively. The visual representation corresponds to information sourced from reality, the practical ground truth, while the simulated representation is its artificial equivalent.

As mentioned in the background section, two key elements need to be included to achieve similarity with the strategies highlighted by psychological research outcomes – some form of simulation (4.2.4) and some probabilistic component (4.2.5) to account for noise in observations and/or predictions.

4.1.1 Environment observation

The top branch of the system serves multiple roles. It is the only part of the system that directly interacts with the external world – in this case, the video data. It must read this data, interpret it and output appropriate information that can be matched to simulation output.

Let V be a video file and w_{video} the output of this branch. The operations described above can be formalised into two functions A_1 (abstraction) and Γ (extraction), such that $w_{video} = \Gamma(A_1(V))$. In this design, the intermediary abstract representation is simply the positions of the objects in the video, as a sequence of coordinates. At implementation time, several methods can be employed to perform this transformation – any tracking algorithm that can correctly follow an object in a series of frames.

A key design decision that arises here is related to the representation of w (i.e. the output domain of Γ), as this will represent the comparison space for the two branches. [6] used a measure of object outline, equivalent to object position and orientation, while [47] performs comparisons in the velocity space – the magnitude of the velocity of an object, mapped against time. The current design uses the latter, as the lack of absolute reference points lifts transformation limitations that would appear between different source environments. This also exploits the properties of the predefined environment: for the same environment, changes in object properties will be reflected in altered velocity profiles and vice-versa – an altered velocity profile will indicate modifications of object characteristics.

4.1.2 Simulation

At an abstract level, the simulation component is responsible for generating a potential outcome just like the human brain would when it tries to imagine the future of a given scene. The simulation will depict the inclined plane scene from the videos, aiming to render the positions of the objects forward in time. Each run is defined by the initial configuration, which essentially instantiates a version of the environment for a given set of parameters – plane inclination angle, initial object positions as well as size, shape, mass and friction coefficients for the two objects. Following the formal description presented earlier, this branch needs to contain an additional simulation function S, which takes into

account the different input type (a set of parameters). If P is this set of parameters, the w_{sim} output of the branch will have the general form $w_{sim} = \Gamma(A_2(S(P))$. Notice how, if the output of the simulation is video frames, A_2 performs the same purpose as A_1. Γ is the same as in the first branch, assuming that A_1 and A_2 have the same codomain.

It is debatable to what degree the accuracy of the simulator, in terms of correctly reproducing physical interactions, affects the performance of the inference pipeline. It is also debatable, with respect to the psychological discussion, whether using a perfect simulator actually represents the imaginative process that the human brain performs, but in the current design this is avoided with the help of noise incurred by the inference strategy.

4.1.3 Probabilistic inference

Adopting the Bayesian perspective, w_{sim} represents the prior belief (hypothesis H), while w_{video} is the newly collected evidence E. The variables whose distributions are of interest here, underlying these beliefs, are the masses and friction coefficients of the two objects. All the other scene/object parameters are considered to be known and can be extracted from video data with auxiliary systems. The expression for the Bayesian posterior distribution update therefore is

$$P(H|E) = \frac{P(E|H)P(H)}{P(E)} \tag{4.1}$$

As $P(E)$ is constant for all E's, the expression becomes $P(H|E) \propto P(E|H)P(H)$. $P(E|H)$ is the likelihood function, indicating how likely the observed data is, given the existing hypothesis.

The goal is to determine the parameter configuration which best approximates the real world scene, by exploring the domain of the input parameters. For this exploration, an appropriate method is the Metropolis Hastings algorithm, which is a form of Markov Chain Monte Carlo approaches. It begins at a random configuration (set of parameters) and, at each step, makes a proposal for a new set of values. This proposal is then stored, thus extending the Markov Chain, or discarded, based on the likelihood of the simulated velocity profile generated by running the simulation with the proposed parameters as input configuration.

Figure 4.2 provides a structural visualisation of this process. For each video, it will be repeated for a set number of times. This is where the overall linear workflow of the system reaches the cyclic phase, looping until a long enough chain is obtained. As this is a Monte Carlo method, a longer chain is expected to provide better results, in a trade-off with time. However, as the chain is randomly initialised, consistent initialisation bias can occur. Usually this is overcome by discarding a proportion of the initial chain steps, corresponding to the so-called burn-in period, but an alternative solution is to run multiple

chains, so that their different initialisations correctly span the proposal space, even at the first step.

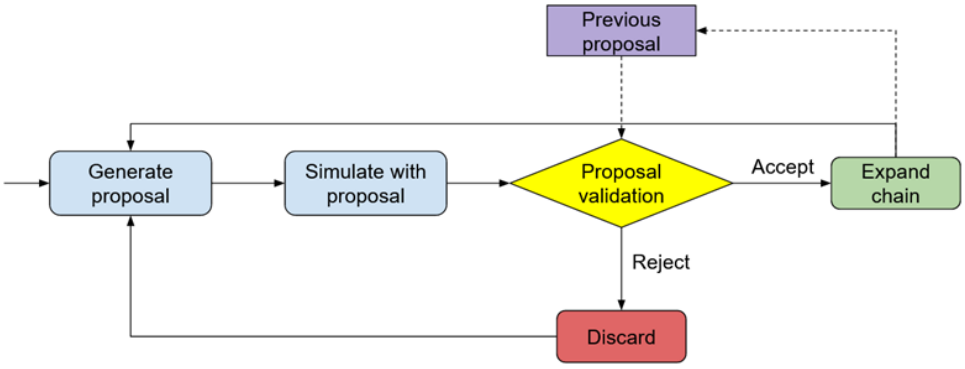


Figure 4.2: Chain expansion workflow

Deciding how to compute the likelihood (involved in the proposal validation) is an important point in the design. Given w_{video} and w_{sim}, the system is in the position to determine how accurate the simulation outcome is, with respect to the observed, real world data. In consequence, a measure of similarity between the two velocity profiles is required. The process of selecting an appropriate method for this will be carefully described in the implementation section (4.2.5), as the final solution was the result of several experiments with the working system.

The MCMC iterations essentially perform a biased sampling of the parameter space, preferring, with a predefined coefficient, the proposals which immediately improve the result, ignoring global progress information. For completeness, this will be compared to optimisation-based methods, such as Stochastic Gradient Descent. However, directly updating the parameter values using a gradient is not attainable with the current architecture, so neural networks will be used in that case.

4.2 Implementation

Having established how the system will operate, at an abstract level, this section will cover the actual implementation in fine detail, touching on more technical aspects. The project was developed in Python 3.9.6 [44] in a Linux-based environment. Specifically, the Windows Subsystem for Linux [26] with Ubuntu 18 [38] was used. This ensured appropriate compatibility with all the necessary dependencies and modules. Python is a general purpose programming language, remarkable for its simplicity and readability. It also offers extensive modules which can perform various tasks, related to specific types of problems. A significant example is the Numpy library [45], a fundamental, widely-used package for scientific computing.

4.2.1 Reading a video

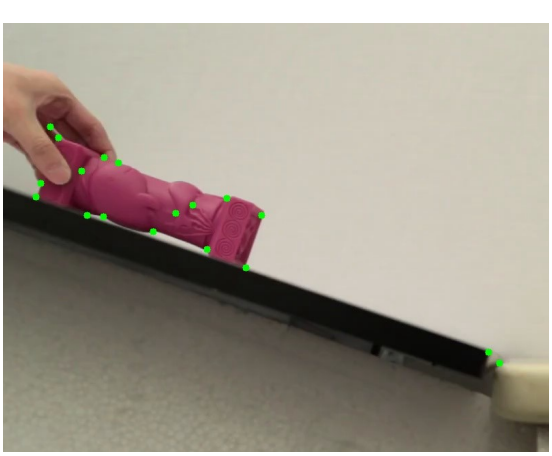
The inference process begins by reading a video from the dataset. For this, the OpenCV library (v4.4.0)[30] is a natural solution, as it provides a straightforward interface, in the form of the `VideoCapture` class. This can be used for multiple types of sources, such as streams or files, to convert them into sequences of frames. In OpenCV, frames are usually interpreted as Numpy multi-dimensional arrays – 2D for grayscale, 3D for RGB – and thus functionality of these two modules can be combined for even more complex operations. The `read()` method of a `VideoCapture` object acts as a Python generator and returns the next frame of the video each time it is called. A loop is thus necessary to iterate through the frames and process them with minimal memory overhead. Video files are identified by their path, so the `VideoCapture` object is initialized with a path as the main argument. Alternatively, in a real-world scenario, this component could utilise a video stream as input, given that a live camera was available for the system.

Once the capture reaches the end of the video file, an empty frame will be returned (the `None` Python type) and the loop is terminated.

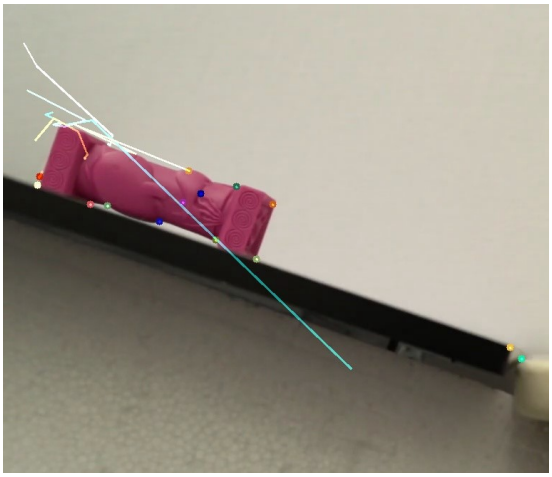
4.2.2 The point tracker

With the purpose of the top branch in mind, the sequence of frames must be interpreted to track the positions of the two objects throughout the video. For this reason, a handy solution is the KLT point tracker method. This is a classic Computer Vision solution that computes registration of a feature (group of pixels) between two images to enable feature tracking by aiming to compute the displacement of this feature. The algorithm was incrementally described and improved by a series of papers [22] [42] [36] published by B. Lucas, T. Kanade, C. Tomasi and J. Shi between 1981 and 1994. But the ability to track does not answer the very legitimate question of "What should be tracked?". Fortunately, this method also offers the necessary logic to determine features appropriate for tracking, such as corners, but this incurs the need for an additional system that, given an initial frame and a set of features, identifies features of interest, i.e. those associated with the free objects in the inclined plane scene.

The OpenCV module provides functionality for Shi-Thomasi feature extraction through the `goodFeaturesToTrack()` method, when run with an appropriate parameter configuration: number of corners to be selected (will be kept in descending order of quality), minimum distance between distinct features, quality score threshold. Figure 4.3a provides an example of a raw set of features found by this method. However, experiments uncovered multiple issues with this approach.



(a) Features identified on the first frame (green)



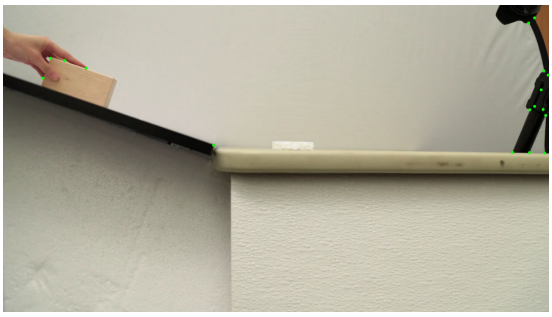
(b) Trajectories obtained by tracking raw features

Figure 4.3: Feature selection analysis - purple rubber

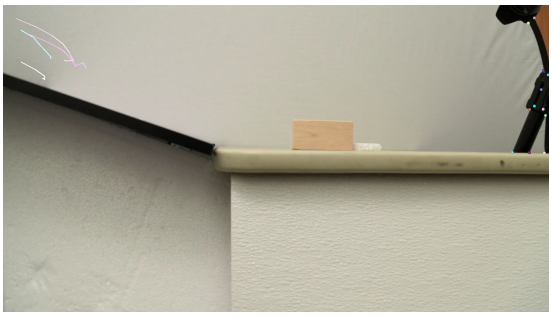
Tracking constraints

First, an obvious problem is the interference of the hand with the objects in the scene. As the hand is usually visible in the first frames of a video, the algorithm will detect some of its features and try to track them. The effect of this behaviour can be observed in figure 4.3b. The resulting traces are noisy and do not accurately represent the movement in the video – the hand simply left the screen, while the purple object remained static.

Another issue with this strategy is that, given how different the objects in the dataset are, a single set of parameters would not suffice in providing a configuration that correctly finds features in all videos. Figure 4.4a illustrates this, by running the same feature detection on a frame depicting a wooden block. Here, as the object of interest has a colour relatively similar to the environment, with no significant contrast, the best features in the image are related to either the hand or the static tripod visible on the right side of the scene.



(a) Features identified on the first frame (green)



(b) Trajectories obtained by tracking raw features

Figure 4.4: Feature selection analysis - wooden block

A potential workaround would be to compute two masks, corresponding to windows around the expected positions of the two objects, such that only features in those areas are stored. But this also has its drawbacks: for the object on the ramp, the hand will most probably often lie in the region of interest, so unwanted features will still be detected. Additionally, given that objects' sizes vary a lot, to maintain generality, one would need to provide a large enough mask, to cover the bigger objects, eventually contradicting the initial aim to improve feature selection.

Labelling as a pre-stage for tracking

All these points lead to a solution that trades generality for performance, by involving a preprocessing stage. In simple words, this would need to consider each video individually in order to determine the features to track, ignoring a potential one-for-all method. The trivial approach that we selected replaces the difficult tuning process of the feature selector with a manual step, in which each video is associated two independent features (two points), one for each object that should be tracked. To achieve this, a separate, external system component was developed, which enables a user to select different points and test the results of the tracking method on those features, to make sure that the output trajectory correctly depicts the movement in the scene. When such points are found, they are stored on disk, to create a Look Up Table-like structure. The workflow is depicted in figure 4.5. During later operations, these points can be retrieved in order to run the tracking algorithm and compute a relevant trajectory.

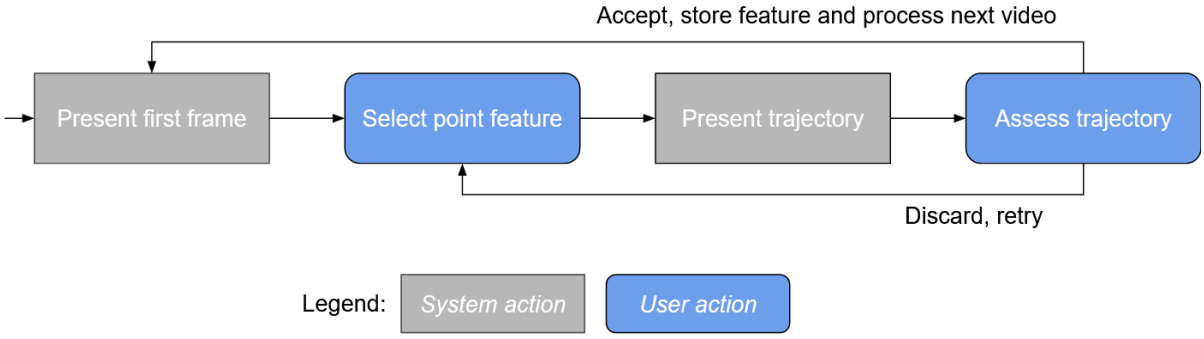


Figure 4.5: Semi-manual feature selection mechanism

In its essence, this operation is equivalent to a labelling stage and somehow alters the nature of the original dataset, in which no such information is stored for each video, while also introducing a hurdle in directly applying this system on an external data source. However, it is important to note that this labelling component can be, in the ideal case, replaced with a dedicated system that performs the same task. We would like to point out background subtraction methods or Deep Learning-based methods which, through an appropriate design, could accomplish this task and thus alleviate the need for manual

intervention. Nonetheless, this would introduce a more complex system, which lies beyond the scope and resources of the current project.

As mentioned earlier, given an initial point, tracking is performed in a KLT fashion, for which an implementation is available in the form of the `calcOpticalFlowPyrLK()` method in the OpenCV module. The arguments for this method are the two frames between which features must be tracked, the position of the features in the first frame, as well as algorithm-specific parameters (the size of the search window, maximum pyramid levels to be used, iterative search termination criteria). These parameters were adjusted through various experiments, until a general solution was found. Tracking is performed in memory, during a single iteration through the video frame sequence, by storing the position of the tracked point(s) and updating it at every frame. The method outputs a series of pixel coordinates describing the trajectory of the object (figure 4.6), which can be interpreted in multiple ways.

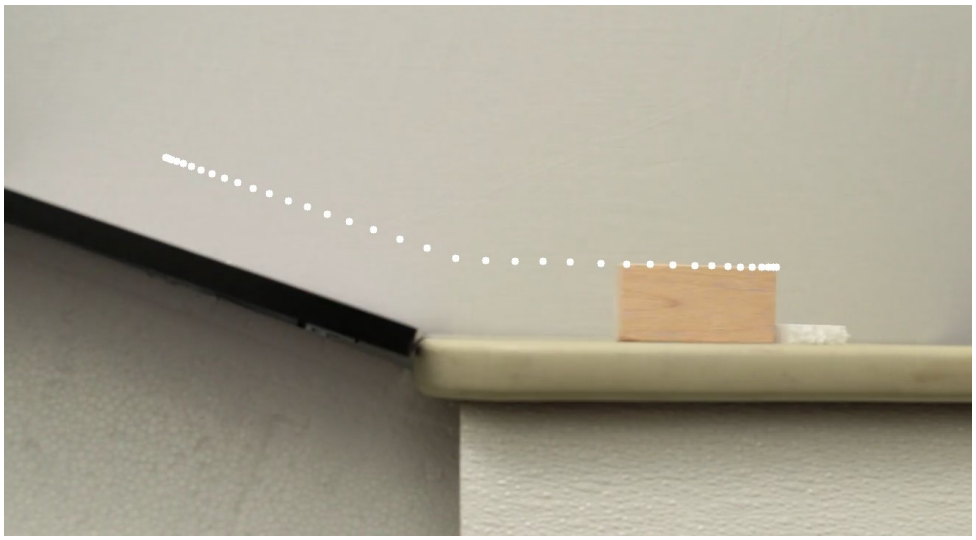


Figure 4.6: Discrete output of tracking algorithm

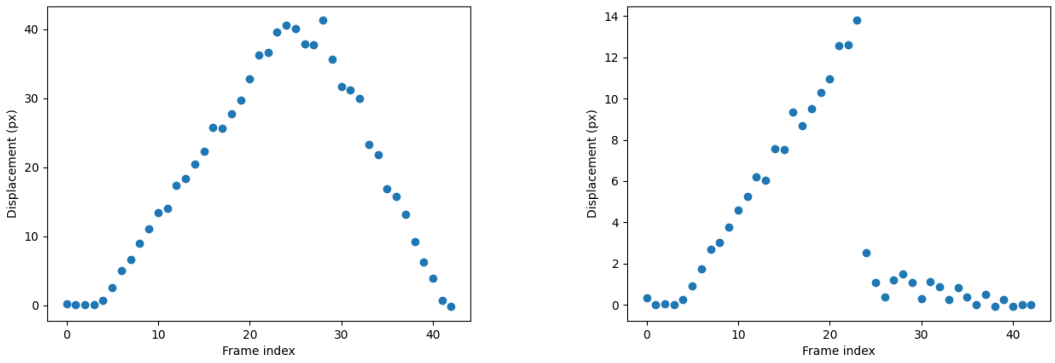
4.2.3 Velocity profile

As specified in the design section, we are interested in the velocity profile – a mapping from time to velocity – so these coordinates must be converted to a velocity measure. Let us consider the physical definition of instantaneous velocity as $v = d/t$. For a video of length N (N frames: $F_1, F_2 \dots F_N$), we want to determine a function $V(k)$ such that for each k in $[1, N)$, $V(k)$ is the instantaneous velocity of the object between frames F_k and F_{k+1}. If the video has frame rate $1/t'$ and recording began at time T, frame k will be captured at time $t_k = T + (k-1)t'$. $V(k)$ will therefore be $\frac{p_{k+1}-p_k}{t_{k+1}-t_k} = \frac{p_{k+1}-p_k}{t'}$, where p_k represents the position of the object in frame k. $V(k)$ will be expressed in pixels/s.

p_k is, essentially, a tuple of two elements, the x and y coordinates of the object, as output by the tracking algorithm. The z dimension is ignored in this case, as it is

assumed to be negligible. In order to preserve axis information, velocity will be computed separately for the two coordinates, so each object will have two velocity profiles, one for each axis. A discussion must also be made for the time reference frame. As the video frame rate is constant, t' can be ignored from initial computations, but velocity will need to be scaled appropriately when compared to data from other sources (e.g. simulation world). With this consideration, velocity becomes simply a measure of displacement, computed independently for the x and y axis. The algorithm presented above converts a video of length N into a velocity profile of length $N - 1$, as the subtraction operation requires two values to generate a result.

The overall aspect of a velocity profile is trivial to describe. For the object on the ramp, as it accelerates by sliding down, the velocity increases linearly towards a maximum speed (reached at the bottom of the ramp), from which it begins to decelerate. For the y axis (vertical), when the object reaches the horizontal plane, velocity immediately drops to 0 (or very close to 0, with some noise). The presence of an obstacle only influences x-axis velocity, incurring an even stronger deceleration. The velocity profiles for the trajectory of the wooden block are depicted as scatter plots in figure 4.7.



(a) x-axis velocity

(b) y-axis velocity

Figure 4.7: Velocity profiles of an object sliding down the inclined plane
As frame rate is constant, displacement (absolute coordinate difference) becomes a measure of velocity.

4.2.4 The physics simulator

Moving further in the system design and looking at the bottom branch, the first important component is the physics simulator. This is a subsystem that should allow for accurate physical simulations of the scene depicted in the videos, such that the velocity profiles extracted from the two worlds can be compared. Simulations can vary in complexity and precision – even plotting the motion of an object with a predefined motion function could represent a simulation – but we are interested in a complete simulation environment that

takes into account gravity as well object interaction (friction, collision).

With the improvement of computing performance, especially that obtained by operations optimised for GPU hardware, physical interactions can now be modelled to a fine level of detail, comparable to ideal mathematical models. Two broad categories of applications that require this are computer games, where realism plays a key role for a pleasant visual experience, and scientific applications – simulations that aid the research process. Monte-Carlo based methods, like the one this project tackles, harvest this ability to run a large number of experiments in exchange for loose initial constraints.

To maintain a uniform code base throughout the project, a physics engine with Python bindings is necessary. Some game engines (e.g. Unity [40]) offer such Python bindings, but there is a discrepancy between our relatively light requirements and the significant overhead that using such a library would incur. As an alternative, the Bullet engine [14] was found to be an appropriate solution. It is an open source project, a library for "*collision detection, rigid body and soft body dynamics*" developed in C++, widely used among robot system developers. A detailed description of how it models physical phenomena is available at [13]. After it is installed using the Python package installer (pip) [33], the `pybullet` module can be imported, enabling the user to design, control and modify a simulation environment from code. Our initial experiments served the purpose of familiarisation with the environment, by running simple simulations with predefined objects and altering basic physical properties, such as gravity or mass, to observe their effects in the simulation world.

Building the objects

The next stage of the implementation process implied a non-trivial method to generate the virtual equivalent of a video scenario in the PyBullet world. Given a video, we would like to extract specific parameters which will be used in setting up the simulation world, such that this becomes an automated process. These parameters can be grouped as follows:

- General: ramp angle, ramp friction coefficient K_i, horizontal plane friction coefficient K_h
- Object 1 (on ramp): shape type, dimensions, mass m_1, friction coefficient k_1, position offset
- Object 2 (on table): shape type, dimensions, mass m_2, friction coefficient k_2, position offset

The mass and friction coefficient of the two objects will act as adjustable input, as these are the variables that are being explored, so they need not be predefined. Additionally, we consider the position offsets to be constant: the first object will start at a fixed height on the ramp, and the second object will start at a fixed y-coordinate on the horizontal

plane. This does not perfectly represent the scenes which will appear in videos, but the effect of these parameters is negligible – as an effect on the velocity profiles, these only translate the scatter along the frame index axis.

Predefined parameters

Another pair of parameters that is fixed is the two friction coefficients of the base surfaces, the inclined and the horizontal plane. In PyBullet, each object material is assigned a surface friction coefficient. When two objects interact, the resulting friction coefficient is the product of the two. As our focus is on the friction coefficients of the objects, K_i and K_h will be set to 1.

For the remaining parameters, multiple approaches are possible. A completely autonomous system would analyse the input video and the associated information (e.g. object trajectory) to extract these parameters – for example, the ramp angle could be computed by determining the slope of the trajectory, for the object sliding down. However, a similar issue to the one identified during the tracking stage arises: the difficulty of designing a global solution. How should the cases where no object movement occurs be tackled? Would the hand interfere with computing object dimensions? We adopted a similar workaround and combined data which can be inferred automatically, such as ramp angle (videos are grouped by this value, therefore this information is pseudo-available in the way data is structured) with a labelling process for object dimensions. This obviously acts as a temporary replacement for a better long-term solution, which would fall under the *future work* spectrum.

Object shapes and sizes

Given the multiple object views that are available, the size of the object can be measured in pixels, for the three axes. For consistent simulations and to maintain the standard unit world that PyBullet uses, this must be converted to metric, by determining a mapping between pixel distance and metric distance. No ground-truth data for the dimensions of the objects is available in the dataset, but some of them, such as trademark plastic blocks Mega Bloks have well-defined dimensions. Based on this, it was established that, when the object is on the left side of the scene, at the top of the ramp, 107.5 pixels in the frame correspond to 6.35 cm in the real world. Distance distortions caused by the position of the object relative to camera lens are ignored, both here and in the tracking phase, when computing the trajectory.

As highlighted when describing the dataset, a wide range of object shapes are present, yet these differences have a very small effect on the output result, so they can be reduced to three general classes which cover the main categories: box – a cuboid shape, defined by 3 dimensions, cylinder – defined by height/length and radius, and torus – 3D ring

defined by the main and secondary radius. At simulation time, each object is associated two components: one of these general classes and a set of dimensions, obtained during labelling.

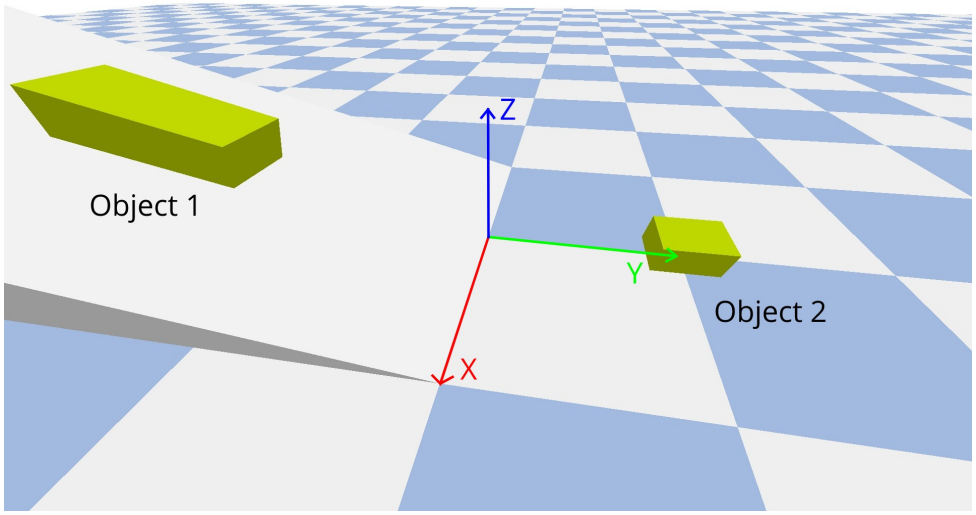


Figure 4.8: Simulation snapshot
Axes and object labels were added separately.

PyBullet provides two operation modes, which can be selected at launch time, depending on use case. These modes differ in whether a Graphical User Interface is available, allowing the user to both observe and directly interact with the simulation world. This was used intensively during setup and debugging phases, to ensure the code generates the expected output (figure 4.8). To add an object to the world using Python, an `.urdf` file is parsed and rendered by the `loadURDF('<filename>.urdf')` method. This is an acronym for "Unified Robot Description Format" [48], used to represent robot models as a set of links with various properties: visual, inertial, collision, contact etc. in an XML format, developed in relation to the ROS environment. Thanks to its widespread use, it was adopted as one of the standard formats for this purpose.

Naturally, any 3D object is a robot consisting of a single link. Inside an URDF file, the geometry of the object can be of four types: cube, cylinder, sphere or mesh – for arbitrary complex shapes. When a mesh is used, it must be specified using another popular file type, `.obj` [15], which describes raw 3D properties of an object, such as vertices, vertex normals, and how vertices connect to create polygon faces. Given a set of scene parameters as detailed earlier, our system must generate custom objects and arrange them in the scene, to reconstruct the visual context.

Building the scene

To begin with, the static context must be initialised – the ramp and the horizontal plane. Despite its apparent simplicity, no appropriate ramp model was available in the PyBullet

object library, so it was designed independently using Blender [12], an open-source 3D creation suite. Two different versions, with 10° and 20° inclination, respectively, were created in this environment and exported as `.obj` files. Then, the physical properties (a very large mass, to prevent the sliding object from displacing it, and a friction coefficient of 1) were defined in an encapsulating `.urdf` file. Similarly, the horizontal plane was custom-defined in order to allow the use of a checker texture, for visual purposes.

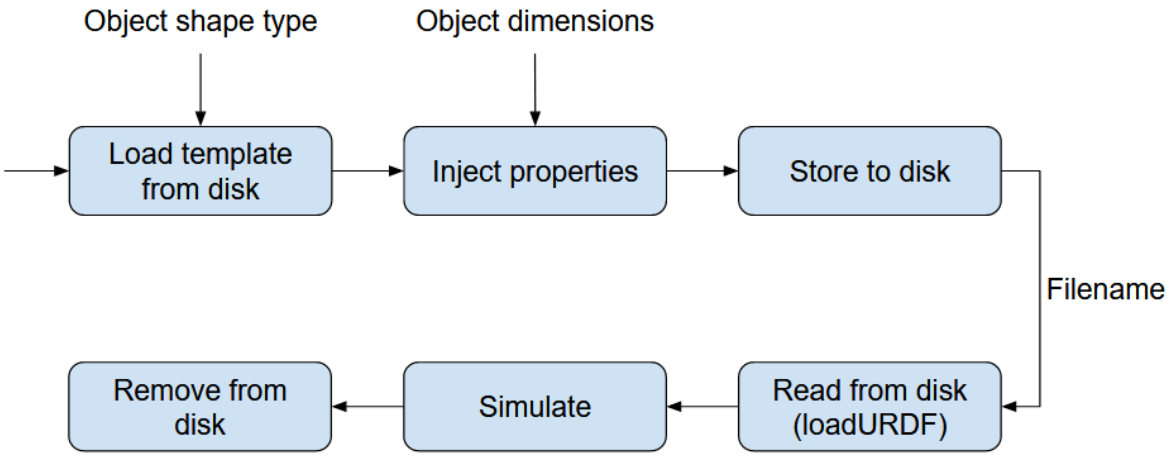


Figure 4.9: Using an object with custom dimensions in a simulation
Disk usage is significant, as the `loadURDF()` method takes input from a file written on disk. A new file is generated each time, as object parameters will be different.

The free objects can only be instantiated at run time, as their definitions depend on the intrinsic physical properties that are being tested, but three templates were designed, one for each category of interest. Cuboid and cylinder classes require no `.obj` file during their creation, because they belong to the basic `.urdf` shapes. Figure 4.9 describes the process of loading such objects. For the torus class, however, a strategy similar to the ramp was adopted – we designed a custom mesh, a torus with main radius 2.5 and secondary radius 0.5. Assuming that torus proportions are constant (or, equivalently, that their effect is negligible), loading a torus object with a custom size involves scaling this predefined mesh by an appropriate factor, by adjusting the `.urdf scale` attribute of the mesh.

Object arrangement

The next step consists of arranging the objects in the scene. By default, the PyBullet simulation environment uses the XY as the horizontal plane, and positive X points towards the user, while positive Y is towards the right. The `loadURDF()` method takes as arguments values for the position and orientation of the base link of a robot, allowing for custom placement. In all scenes, the ramp has a fixed position: the inclined edges are parallel to the plane determined by the YZ axes, the lower side of the ramp is towards positive Y, with the middle of this edge lying at the origin of the coordinate system, as

can be observed in figure 4.8. Any other orientation could have been used, but this one was selected for visual consistency with the video scenario that is being analysed.

Out of the two moving objects, the second one is easier to add to the environment. Its (x_2, y_2, z_2) starting position tuple will be $(0, Y_{offset_2}, d_{z_2}/2)$, where Y_{offset_2} determines how far this object is from the base of the ramp, and d_{z_2} is the height of the object (the z dimension). For the object on the ramp, however, some additional computation is required, to take into account the ramp inclination. The z coordinate depends on Y_{offset_1}, as such:

$$z_1 = |Y_{offset_1}| \cdot \tan\theta + d_{z_1} \frac{1}{\cos\theta} \tag{4.2}$$

implying that $(x_1, y_1, z_1) = (0, Y_{offset_1}, z_1)$. A graphical interpretation is available in figure 4.10. The roll-pitch-yaw orientation tuples for the two objects will be $(-\theta, 0, 0)$ and $(0, 0, 0)$ respectively.

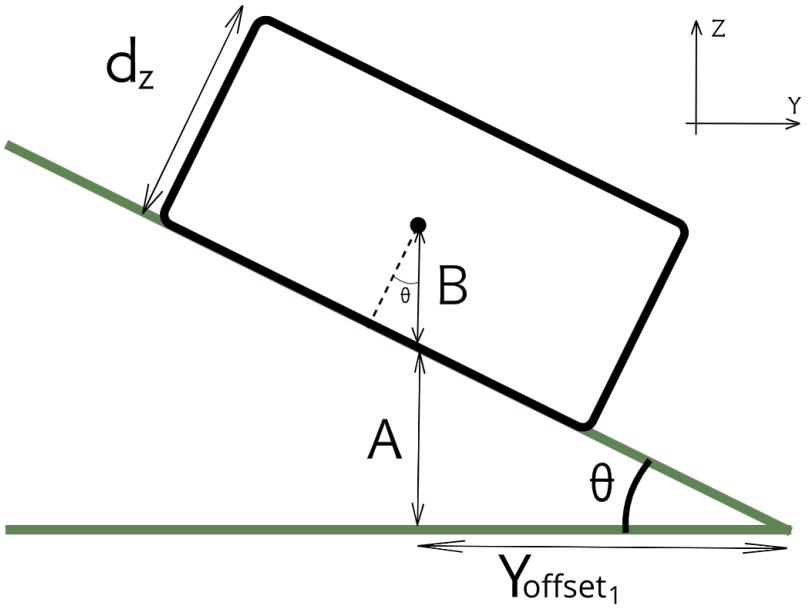


Figure 4.10: Computing the position of the first object on the ramp
The z component is $A + B$ = height of ramp at Y_{offset_1} + angular height of object. See equation 4.2.

Running simulations

After scene setup is complete, the simulation process can begin. The core functionality here is provided by the `stepSimulation()` method of the `pybullet` module. This function effectively advances the simulation scene along a time axis, with a discrete predefined time step (1/240 s). This value can be altered, but such modification is not recommended, as several underlying computations depend on it and are optimised accordingly. As the simulation advances, object positions are queried (`getBasePositionAndOrientation()`) and stored, in order to obtain the equivalent of the trajectory extracted from video. We

note that the simulation medium offers a much better granularity (240Hz vs. 29.97Hz ≈ 30Hz), uncovering the need for a method which bridges this gap and solves time correspondence problem between the two media. This falls back to the velocity profile definition (section 4.2.3) and the importance of t'. Unless an interpolation algorithm is used, 30Hz, the video framerate, is the upper bound for the density of the velocity profile. In consequence, the simulation data must be resampled at this frequency. The simplest solution is to only store the positions of the objects at simulation steps which have a corresponding frame at that timestamp, as seen in figure 4.11.

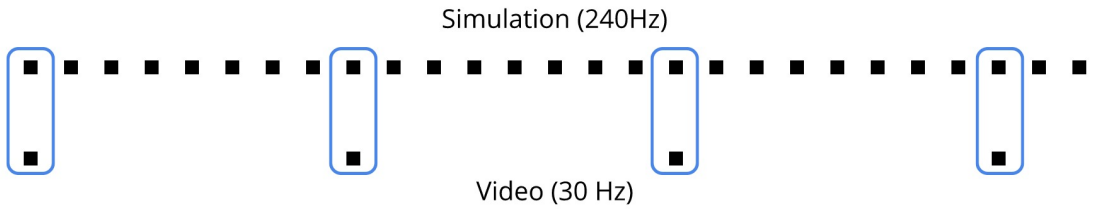


Figure 4.11: Pairing simulation steps with video frames
There are approximately 8 simulation steps per video frame. The pairs grouped by the blue outlines are kept. The remaining simulation step data is discarded (steps are skipped).

A detail that might interfere with this mechanism is the fact that videos can begin at arbitrary time points, with respect to when the object is actually released, due to potential lack of coordination between video capture and the hand letting go of the object. In comparison, the simulation always begins with a completely free object, ready to slide down the ramp. This problem is, however, very difficult to alleviate and as its effect is minimal, it will be neglected.

The other dimension that needs to be correlated is the distance travelled by an object between two consecutive frames/simulation steps. To achieve this, the pixel-to-metric conversion step that was used during object generation is applied on each value of the velocity profile extracted from video. The final velocity profiles will therefore express displacement in meters, sampled at a frequency of 30Hz. For each object, two axes will be considered: X, the horizontal video axis, equivalent to the simulation Y axis, and Y, the vertical video axis, equivalent to the simulation Z axis. During execution, all simulations will run for a fixed number of steps, determined by the size of the longest video in the dataset. Setting this constraint enables easier operation on the velocity profiles, later on.

Experiments with object parameters

Before addressing the inference logic of the pipeline, we would like to perform a simple analysis on the velocity profiles extracted from simulations, in order to better understand the relationship between the intrinsic object parameters and the resulting profiles. For

all experiments, four graphs are presented, showcasing the four velocity profiles in two parallel scenarios, between which a single system parameter was altered.

The first experiment looks at the influence of the mass of the first object (m_1). In experiment 1A, $m_1^A = 1$, while in experiment 1B, $m_1^B = 0.3$, and $m_2^{A,B} = 0.5$. The profiles are identical up to the collision point, but a higher mass incurs a smaller deceleration on the first object as well as a higher energy gain of the second object, observable in the amplitude of the X2 velocity profile.

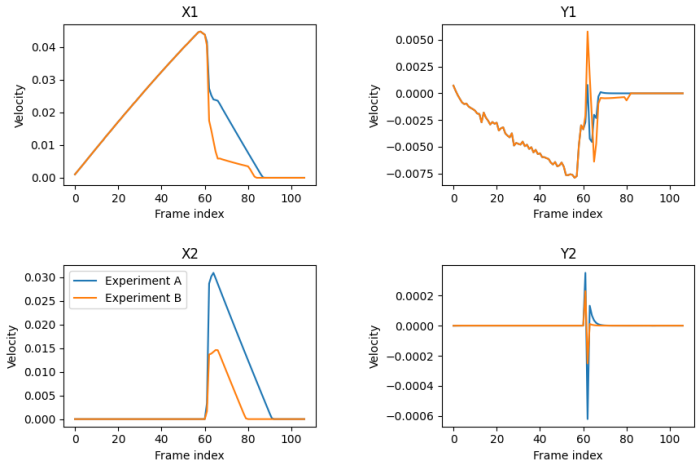


Figure 4.12: Experiment 1. All parameters are identical, except for m_1
X1, Y1 and X2, Y2 refer to the velocity profile for the corresponding axis, for object 1 and 2 respectively. Parameters for experiments A and B are (1, 0.1, 0.5, 0.1) and (0.3, 0.1, 0.5, 0.1) respectively, and a 10° ramp is used.

In the second experiment, the value that changes is k_1: 0.2 for experiment A and 0.1 for experiment B. The outcome difference is that, in B, both objects reach a higher velocity and object 1 has a higher acceleration prior to the collision (the slope of the velocity profile).

Interestingly, but backed by physical explanations, experiment 3 yields results that are similar to experiment 1, as it is m_2 that changes. $m_2^A = 1, m_2^B = 0.3, m_1^{A,B} = 0.5$. This highlights how, for identical friction coefficients, it is the relation between m_1 and m_2 that determines how the graphs look. The masses tuple (1, 0.5) from experiment 1A leads to a result similar to the tuple (0.5, 0.3) from experiment 3B. In both cases, $m_1 > m_2$.

The final experiment looks at the influence of k_2. A clear feature determined by this parameter is the acceleration of the second object after the collision. When k_2 is bigger, the object will slow down noticeably faster. It is also worth pointing out that the collision might not be necessarily be a simple/single interaction between the two objects, as it can be deduced from the X1 velocity profile of the experiment.

As observed, the trivial physical system that we are considering offers a wide complexity of phenomena whose effects can be traced to the underlying properties of the objects involved. To explore this trace and recover these properties, the following probabilistic

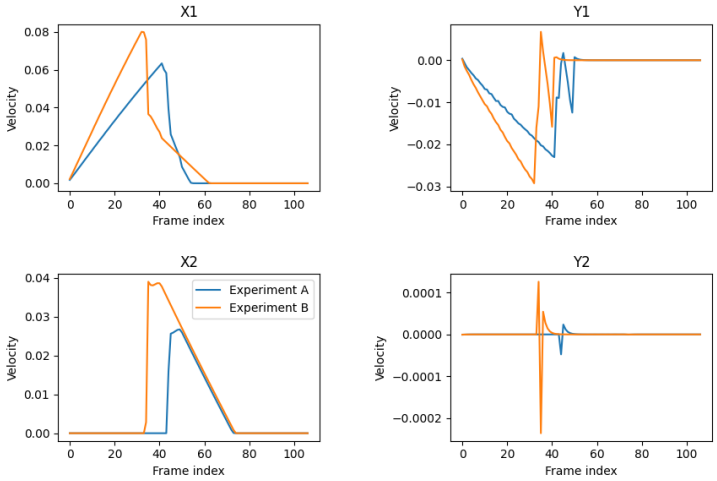


Figure 4.13: Experiment 2. All parameters are identical, except for k_1 Parameters for experiments A and B are (0.5, 0.2, 0.5, 0.1) and (0.5, 0.1, 0.5, 0.1) respectively, and a 20° ramp is used.

method was adopted.

4.2.5 MCMC exploration

The name *Markov Chain Monte Carlo* represents a very broad category of algorithms with a random component in which a Markov chain structure is used. The specific variant that this system uses, Metropolis-Hastings, introduces the notion of a relative proposal distribution Q, with respect to the current Markov chain step X_t. Such a step consists of a tuple of values, one for each of the parameters of interest: $X_t = (m_1, k_1, m_2, k_2)$. Formally, this tuple will be referred to as X_t, while individual elements in the tuple are x_t. Then, a candidate next step X'_{t+1} is sampled from Q, and a decision must be made whether this is accepted or discarded.

For simplicity, a uniform distribution with different widths, centered at x_t is used – a distinct distribution for each component. One implementation of this behaviour is available in figure 4.16. From this point of view, the chain is independent across the four dimensions, yet components are overall correlated by the candidate acceptance procedure. Given X_t, X'_{t+1}, a simulation function S and a likelihood function L, a decision variable is defined as $\alpha = \frac{L(S(1))}{L(S(2))}$. If $\alpha \geq 1$, the proposal is saved and added to the Markov chain. This expresses the fact that the proposal is closer to the observed, real-world scenario, and should definitely be stored. On the other hand, when $\alpha < 1$, the decision to save X'_{t+1} can take multiple forms.

One example involves using alpha as the acceptance probability: draw a sample p from a uniform (0, 1) distribution; if $p < \alpha$, accept, else reject. However, this method indirectly depends on the likelihood computation, and experiments showed that in our use case it would result in very few samples being rejected. An alternative that avoids this involves

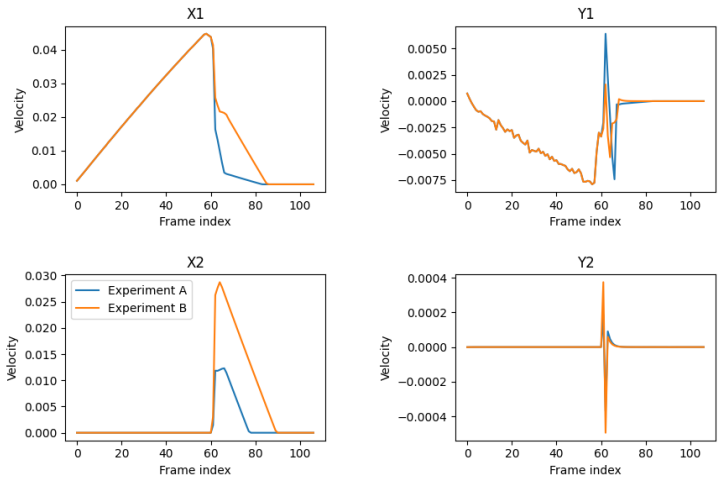


Figure 4.14: Experiment 3. All parameters are identical, except for m_2 Parameters for experiments A and B are (0.5, 0.1, 1, 0.1) and (0.5, 0.1, 0.3, 0.1) respectively, and a 10° ramp is used.

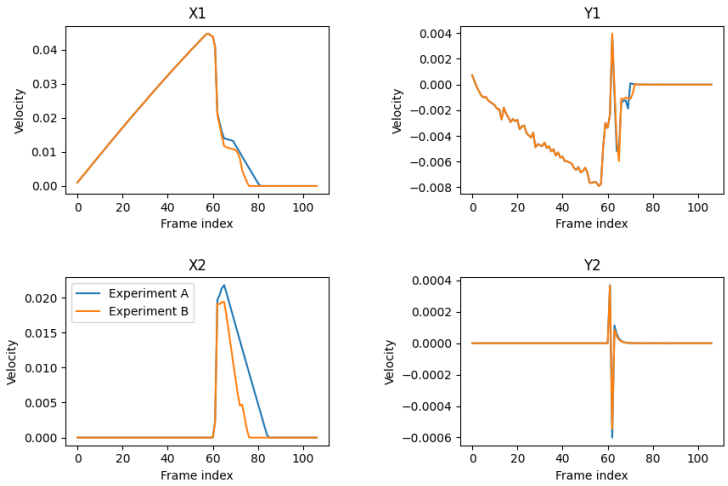


Figure 4.15: Experiment 4. All parameters are identical, except for m_2 Parameters for experiments A and B are (0.5, 0.1, 0.5, 0.1) and (0.5, 0.1, 0.5, 0.2) respectively, and a 10° ramp is used.

an additional independent decision at this point: accept X'_{t+1} with a probability of 50%, equivalent to a coin toss. Note that in a strict optimisation scenario, proposals with $\alpha < 1$ could be discarded straight away. This results in a naive random optimisation algorithm, which is unable to avoid local maxima (with respect to the likelihood) in the parameter space which are further away than the proposal distribution range.

For the rejection case, to ensure a constant chain length across multiple runs, the previously accepted value (i.e. X_t) can be re-added to the chain, but this has the drawback of altering aggregate chain measures, as it introduces bias (some values will appear multiple times, just because no adequate proposal was found).

The complete algorithm, in pseudocode form, is listed as Algorithm 1. For each video, 16 such chains are run, each consisting of 75 proposal steps. These values can be adjusted

```python
def get_proposal(x, width):
    delta = np.random.rand() * 2 - 1
    x_new = x + delta * width

    while x_new <= 0 or x_new >= 1:
        delta = np.random.rand() * 2 - 1
        x_new = x + delta * width

    return x_new

def get_proposals(variables, width):
    return [get_proposal(x, width) for x in variables]
```

Figure 4.16: Code snippet - sampling from a proposal distribution of a certain width The `numpy.random.rand()` method is used for uniform sampling, and boundary checks are performed: all resulting values should be between 0 and 1.

freely and they essentially represent a trade-off between running time and output quality. As this is a Monte Carlo method, a longer chain is expected to yield better results, but this increases the time needed for the inference process to finish. As it can be observed, the width of the proposal distribution is 0.05 for most steps, but it raises to 0.5 once every 20 steps, to allow for greater variation in the exploration process. It is important to note that the chains advance simultaneously for all four parameters. An incremental solution, which identifies the best value for a single parameter at a time, is not appropriate in this context, as their effects are interdependent.

The likelihood function

It has not yet been clarified how the similarity between two velocity profiles is calculated. In fact, this task refers to multiple such sequences: each video/simulation processing component outputs four velocity profiles of length T, grouped as two array objects of size $[2, T]$. In preparation for the comparison step, these are concatenated based on axis (Object 1 X + Object 2 X, Object 1 Y + Object 2 Y), then joined to form a 1D array. T can vary based on video duration, and simulation output will be trimmed to this length. Ultimately, comparison occurs between two sequences of size $4T$, named A and B for convenience.

There are plentiful methods which suit the task of measuring similarity. After all, any error function can be converted into a likelihood value by inverting it. We have looked at three independent methods: Gaussian similarity, Earth mover's distance (EMD) and Root Mean Similarity (RMS), which were compared formally using a custom framework.

Gaussian similarity refers to computing the conditional probability between the two vectors. This is performed element-wise, as follows. Given an element $A[i]$, we consider

Algorithm 1: Metropolis Hastings with MCMC for parameter exploration

Input: A path to a video file, number of steps N
Output: A list of tuples representing values of (m_1, k_1, m_2, k_2)
V_{obs} = velocity profile from video;
output_chain = [·];
Draw $X_{current} = (m_1, k_1, m_2, k_2)$ at random;
V_{sim} = velocity profile when simulating with $X_{current}$;
$L_{current}$ = likelihood of V_{obs} given V_{sim};
for $i \leftarrow 1$ **to** N **do**
    width = 0.05;
    if *i*%20=*0* **then**
        width = 0.5;
    end
    Draw $X' = (m_1', k_1', m_2', k_2')$ using $X_{current}$ and *width*;
    V_{sim}' = velocity profile when simulating with X';
    L' = likelihood of V_{obs} given V_{sim}';
    $\alpha = L'/L_{current}$;
    save = `True`;
    if $\alpha < 1$ **then**
        r = random variable in (0,1);
        if $r < 0.5$ **then**
            save =`False`;
        end
    end
    if *save* **then**
        $L_{current} = L'$;
        Append X' to *output_chain*;
        $X_{current} = X'$
    end
end

the normal distribution N_i with mean $A[i]$ and variance k (k has a negligible effect on the outcome; it was set to 0.05). Let s_i be the value of the probability distribution function of N_i evaluated at $B[i]$. We then define

$$\text{Gaussian similarity} = \sum_{i=1}^{4T} \log s_i \tag{4.3}$$

This is a similarity measure because the result will have a maximum value when the two velocity profiles overlap, as the mode of the distribution will be selected at each discrete element index. In our implementation, this method uses the `norm` class of the `scipy.stats` module, and is optimised using vector operations from NumPy (e.g. `log`).

A second similarity metric being analysed is derived from Earth Mover's Distance, which allegorically refers to the minimum effort needed to transform a pile of dirt such that its shape resembles another pile. For two vectors, this can be computed using a dynamic programming approach, using a 2D array M initialised with a maximum distance value. $M[x][y]$ refers to the minimum cost between sequences $A[1], A[2], ...A[x]$ and $B[1], B[2], ...B[y]$, so when the matrix is full, $D = M[4T][4T]$ will correspond to the cost for the entire sequences. $M[0][0]$ is initialised to 0. Then, given a distance function $d(\cdot, \cdot)$, M is filled sequentially by setting $M[i][j] = d(A[i-1], [B[j-1]) + min(M[i -$

$1][j], M[i][j-1], M[i-1][j-1])$. Finally, to convert the distance to a similarity measure, $1–D$ is returned. It was empirically determined that $D < 1$, implying that this is a valid operation.

Finally, another alternative for likelihood computation is a modified version of the Root Mean Squared Error, from which the squaring operation was eliminated, and a subtraction from 1 was added, to transform the error into a similarity value. For comparing elements in pairs, the distance function $d(x, y) = 1 - \frac{|x-y|}{|x|+|y|}$ is defined. Then, these errors are averaged and the square root is returned.

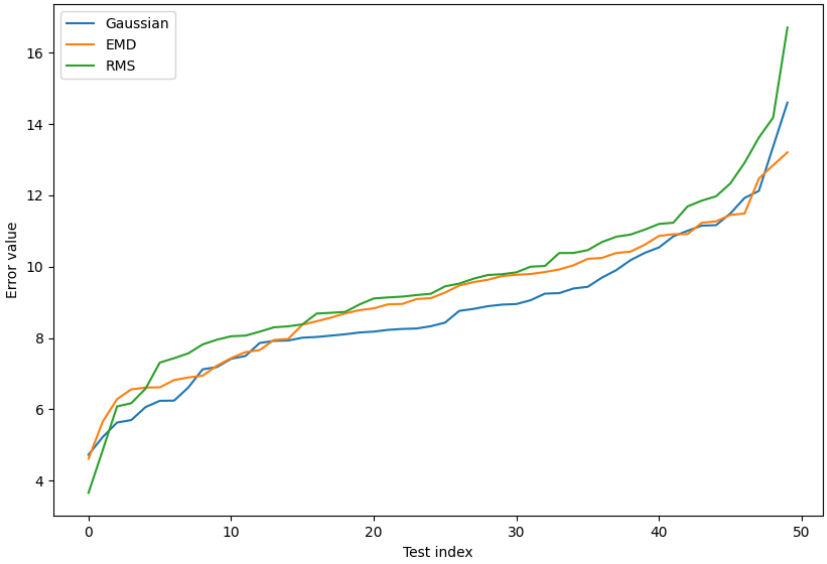


Figure 4.17: Analysis of 3 similarity methods across 50 chains
The value corresponds to the error between the histogram and the equivalent Gaussian distribution, computed at bin centres.

Likelihood function evaluation

Comparing these three methods starts with the underlying assumption that, as the MCMC chain expands, the generated samples should form a Gaussian distribution. This reasoning is based on the fact that Normal distributions are the most natural representation of a belief and they appear in arbitrary scenarios in the technical and not-so-technical world. Given a chain, we aim to determine how close it is to a Normal distribution. This is achieved in three stages. First, the samples from the chain are converted into a histogram of 50 bins, over the $(0, 1)$ interval. During this entire process, only the k_1 value is considered – it can be argued that it has the highest influence on the velocity profile. Then, given the mean and the variance of the values in the chain, the PDF of a corresponding Normal distribution can be determined. Finally, for each bin of the histogram, we compute the error as the L1 distance between the bin value and the equivalent evaluation of the PDF at the centre of the bin, returning the sum of these distances.

Overall, for each of the three methods, 50 chains of length 75 are analysed. The errors were sorted in ascending order and plotted, revealing that the Gaussian similarity computation has the smallest error, on average, at a close margin from the others (figure 4.17). Therefore, this was selected as the primary likelihood computation method.

Selecting a result

Figure 4.18 shows the progress of a MCMC exploration and associated values.

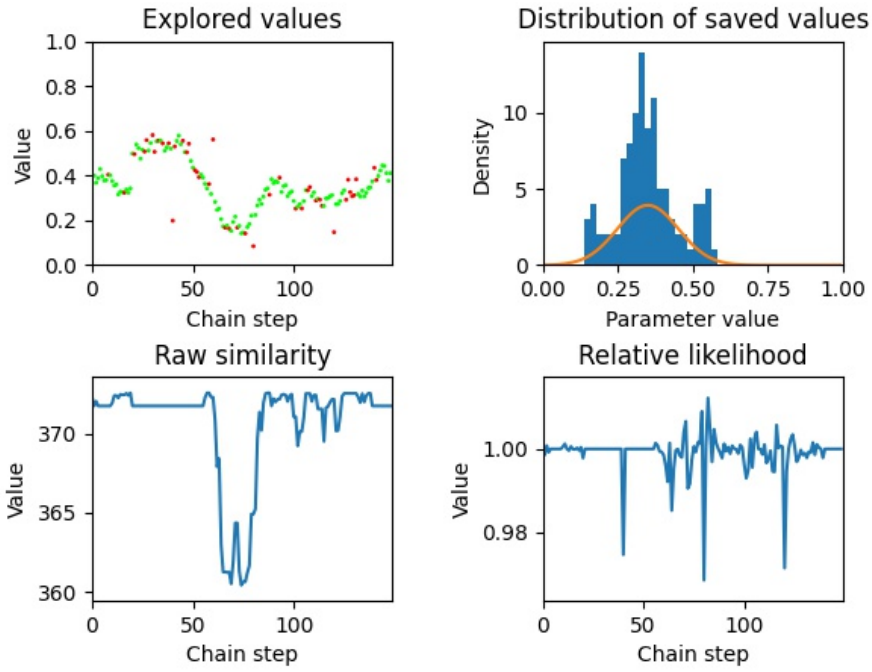
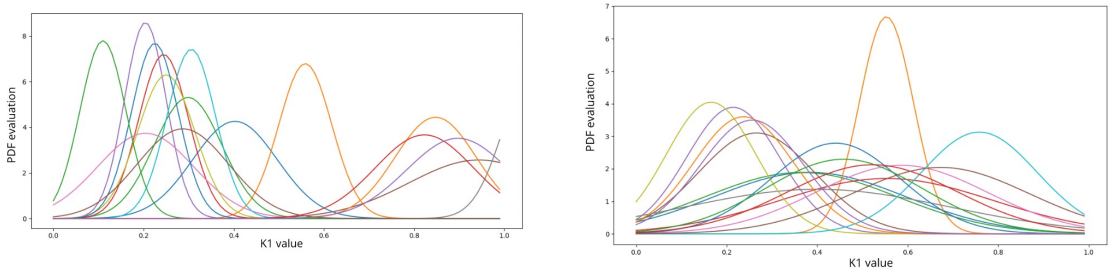


Figure 4.18: The exploration process using MCMC Metropolis-Hastings
Top left: raw explored values for the k_1 parameter. Saved proposals are in green, discarded proposals in red. **Top right:** the histogram and sampling distribution of the saved values for k_1. **Bottom left:** the raw similarity value of each step (involves all 4 parameters, m_1, k_1, m_2, k_2). **Bottom right:** the relative likelihood of each step, with respect to the previous step - essentially, the α decision variable.

One more key decision must be made, in order for the system to be complete and, for a given video, to return a prediction for the four parameters. Following the Bayesian framework, one proposed solution was to treat each chain (out of the 16 ran for a single video) as a Normal sampling distribution, and store the chain with the smallest variance, as this would imply highest confidence in the explored values. However, some experiments showed that low variance can also be caused by initialisation and exploration bias, leading to very confident outliers, far from the correct value. Figure 4.19a shows a case where the majority of the 16 chains are correct (the expected output is somewhere around 0.2), and the chain with the highest mode/lowest variance is an appropriate prediction. In

figure 4.19b we present a case where an outlier with abnormal confidence would lead to an incorrect prediction.



(a) Correct case, distribution with highest mode is a correct prediction.

(b) Incorrect case, distribution with highest mode is a biased outlier.

Figure 4.19: Sampling distributions obtained from 16 chains.

The alternative solution involves looking at independent steps in the chains, instead of analysing chains as a whole, and selecting the one which came as close to the observed phenomenon as possible. Essentially, the proposal with the highest similarity is kept. This is grounded in the human ability to identify local optimal solutions even before the entire search space has been traversed. It comes natural to us to remember a specific instance, a singular element that stood out from a larger set thanks to a particular feature or behaviour. Translated to the current scenario, the elements being analysed are independent simulations of the observed scene. When such a simulation performs very close to the real-world example, it will be stored, until potentially a better one is found.

4.3 Evaluation

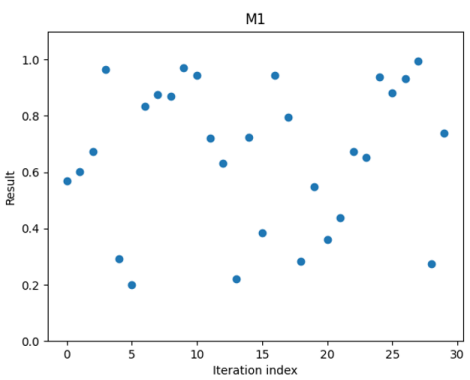
Throughout the implementation stage, a test-driven approach was used, so the need for a dedicated testing phase is lifted. Nonetheless, we would like to evaluate the system with respect to how well it fulfils its overarching goal and the requirements defined in the beginning.

Given that no ground truth parameter information is available for the selected dataset, it is impossible, in the current setup, to perform an analytical evaluation on the accuracy of the outputs. However, there still exist several methods which provide useful insight about the degree to which the system *understands* the world.

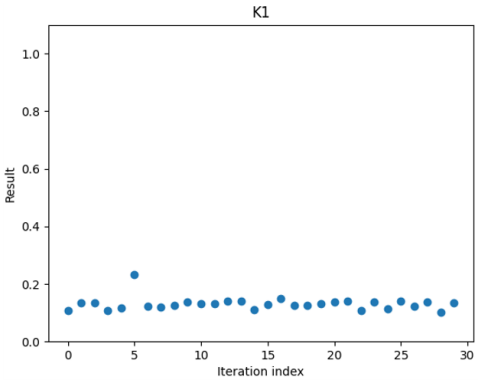
Precision of predictions

The first one refers to assessing the precision of the system. In contrast with accuracy, which considers how close a value is to the actual expected value, precision focuses on how close values are to each other, for the same item being analysed. If

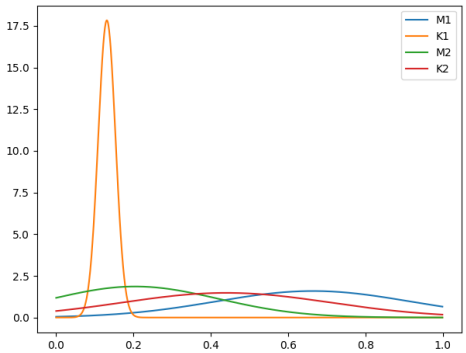
$P = [(m_1^1, k_1^1, m_2^1, k_2^1), \ldots (m_1^n, k_1^n m_2^n, k_2^n)]$ is a list of tuples obtained as result when running the algorithm n times on the same video, a measure for precision is the variance, computed independently for each of the parameters. For example, for parameter m_1, if the mean across m_1 is denoted $\bar{m_1}$, the variance will be $\sum_{i=1}^{n} \frac{m_1^i - \bar{m_1}}{n}$.



(a) Scatter of m_1 values across 30 runs.



(b) Scatter of k_1 values across 30 runs,



(c) The Normal distributions associated with the values obtained for each parameter, across 30 runs.

Figure 4.20: Precision evaluation

The outcome of such an experiment can be visualised in various ways. Figures 4.20a and 4.20b show the results as a scatter of values, for the mass and friction coefficient of the first object, respectively. Figure 4.20c considers all four parameters and plots the Normal distributions which have the same mean and standard deviation as the corresponding set of values. This uncovers a very interesting shape of the results. Clearly, the system performs very well in predicting the value of k_1. Yet, for the other parameters, the outputs vary by a high margin, indicating a very erroneous behaviour.

An explanation of this behaviour intuitively lies in the relationship between the parameters, the velocity profile and the likelihood function. Surely, changing even a single parameter alters the shape of the profile, but parameter k_1 naturally has the highest influence in this sense. Considering mainly the X-axis profile for Object 1, changes to the velocity slope during the initial sliding phase have a critical effect on the rest of the

profile. From a mathematical point of view, this is equivalent to a partial derivative of the function which generates the velocity profile, with respect to each of the parameters – for k_1, this partial derivative would have the highest value among the four. The likelihood function is prone to accentuate this difference, as each of the discrete points in the velocity profile has equal contribution to the overall result. The parameter search is not directly penalised for wrong proposals, but a wrong proposal for k_1 will, for the reasons described above, have a higher chance of leading to a lower likelihood value (than a wrong proposal for m_2, for example), implicitly incurring a higher chance of the proposal tuple being discarded.

A hypothetical workaround is to determine a comparison space (not the velocity space) in which the four parameters are equally weighted, or include weighting information about the parameters in the likelihood function, such that the process is not biased towards any of these. Nonetheless, it is important to point out that, in the context of a framework that imitates human behaviour, we lack a measure of how well humans can approximate intrinsic physical properties. It can be argued that humans focus on the effects of properties in a relational manner – for example, is this object heavier than the other? It is usually questions like this that provide the basis for useful behaviour prediction, rather than explicit values.

For the corner case of videos depicting no movement, due to a high friction coefficient or low ramp angle, there is clearly no information to be extracted from velocity data. In this situation, the system correctly outputs a set of parameters that generate a static scene, such that the velocity profiles match, but the actual value for the parameters has no explicit meaning - the domain of solutions is infinite.

Comparison with a naive guesser

A second evaluation method compares the current system with the naivest of implementations, to check against the lower bound of the range of solutions. Instead of a complex mechanism for inferring the parameters of interest, they are simply sampled independently, with random probability, from the predefined domain (0, 1). Then, to verify the accuracy of such a prediction, a straightforward process is followed. A simulation with that set of parameters is run, and an accuracy measurement, in the form of L1 distance from the equivalent velocity profile extracted from video, is computed. Averaging this distance over 30 experiments yields an error margin of 0.78 for the randomly sampled parameters, while the error margin for the parameters inferred through MCMC with simulation is 0.43. If, instead, L2 norm is used, these values become 0.11 and 0.07 respectively. This clearly indicates that our system is a valid solution that brings improvement over a simple guess.

Simple ablation study for Metropolis Hastings

Thirdly, we would like to assess the importance of how samples are saved/discarded during the MCMC chain expansion step. For this, the system is compared to a similar architecture, which does not involve likelihood measures in the decision step. Therefore, proposals are stored based on a random decision, equivalent to a coin toss. The same distance metrics were averaged over 30 experiments, resulting in 0.81 with L1 and 0.11 with L2. This experiment simultaneously supports two ideas: it is an empirical proof for the improvement that a Metropolis-Hastings method generally brings over a simple random walk exploration, while also demonstrating its usefulness in this particular case.

Visual evaluation

A less formal evaluation method consists of analysing data extracted from the physics engine, in the form of simulation scene snapshots. This enables a qualitative analysis of the effects of the parameters. By comparing the last frame of the video and the equivalent frame from the simulation world, we trivially assess how close the output of the system is to reality. An example is available in figure 4.21. At a high level, the pairs are very similar, yet not perfectly identical, usually with an error that would be negligible for a human agent.

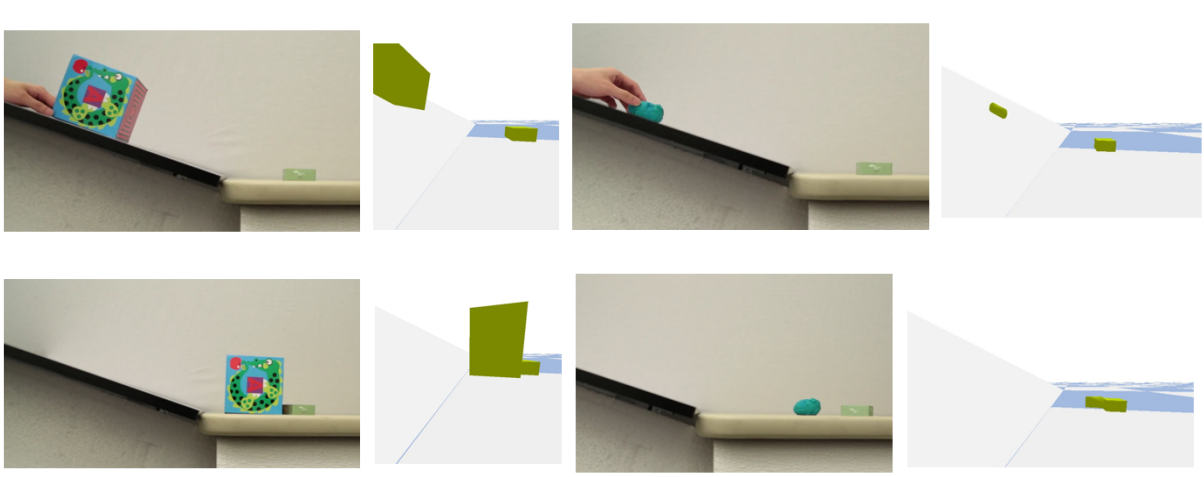


Figure 4.21: Frame comparison
Top row: first video/simulation frame, next to each other. **Bottom row:** last video/simulation frame.

Potential tests

Coming back to the applicative side of predicting object movement, Wu et al. [47] proposed two interesting tests for a system of this kind. However, both of them require a human benchmark for comparison, which lies beyond the time and work resources of this project. The tests aim to put the system head-to-head with a human counterpart, on two

vital questions: "Will the object on the inclined plane slide?" and "How far will it slide?". For the first test, a simple yes/no answer can be output by the system, after comparing the predicted friction coefficient to the maximum friction coefficient that triggers a sliding behaviour (this can be derived mathematically as a function of the angle of the ramp, as explained in section 3.1.1). The second question involves simulating the scene until all movement halts, then retrieving the positions of the objects, to generate a prediction which is compared to the one sourced from the human experiment. Their results show that, on average, the system performs as good as humans, and we are confident that our system can achieve similar performance, if tested in the same framework.

Finally, for completeness, it is important to mention that the current system successfully verifies all the requirements defined in section 3.2, as they underpin the main design decisions.

Chapter 5

The Neural Network model

5.1 Overview

This chapter focuses on the second part of the project, which can be perceived as either an independent pursuit of the same goal – extracting physical properties from video – or an extension of the first part, which distances itself from the particular method that was initially targeted and concentrates on the quantitative side of the results.

The hypothesis that silently underlies our work is that the velocity profiles of the two objects provide sufficient information for extracting the four parameters of interest. The Bayesian model harvests this information indirectly, during the chain expansion process, but the search is just loosely guided by the likelihood function involved in proposal acceptance. Instead, we would like to explore the possibility of directly analysing the velocity profile in order to determine appropriate values for the parameters, using an approach based on Machine Learning (ML). Formally, this is equivalent to finding a mapping $f : S_{VP} \to \mathbb{R}^4$ from the set of velocity profiles S_{VP} to the set of parameter tuples, such that, if $f(w) = (a, b, c, d)$, a scene whose objects have free parameters (a, b, c, d) and a fixed set of parameters T (representing general scene parameters, ramp angle, starting positions for objects, etc.), the velocity profiles of the objects in the scene will be w.

As it can be noticed, this approach involves a shift in the character of the problem. If, in the Bayesian model, parameters were determined by analysing each video individually at prediction time, with the inference process happening live, a ML model will operate differently – the training stage is the most time-costly, during which the mapping is learned offline. A prediction will then simply consist of running a velocity profile obtained from video through the model.

5.2 The dataset

Naturally, training the model requires an extensive amount of labelled data, with ground truth values for the output components, so the Physics 101 dataset [46] is not a feasible solution. However, the physics engine used earlier is a great tool for this task. Following the definition of f, we are interested in input-output tuples where the input component contains the velocity profiles of the objects in the scene, and the output data consists of the m_1, k_1, m_2, k_2 values.

To generate this, a reverse strategy is applied: first, a set of parameters is sampled at random. This selection is constrained to a custom interval for each of the parameters – masses will be chosen from $(0, 1)$, while friction coefficients from $(0, 0.3)$. Although small, these ranges provide a challenging search space, with many interaction types, depending on the relation between m_1 and m_2. The friction coefficients constraints ensure that the first object will slide. Next, a predefined scene is simulated with these parameters. We call the scene "predefined", because all other arguments are fixed: the sizes of the two objects, their initial positions and the ramp angle (20°). A snapshot obtained during this phase is available in figure 5.1. The simulation runs for 60 steps, at a frequency of 30Hz, and the velocity profiles of the two objects (four, in total) are processed and stored in a file. Each entry in the dataset will ultimately comprise one input and one output array of sizes (4, 60) and (1, 4) respectively.

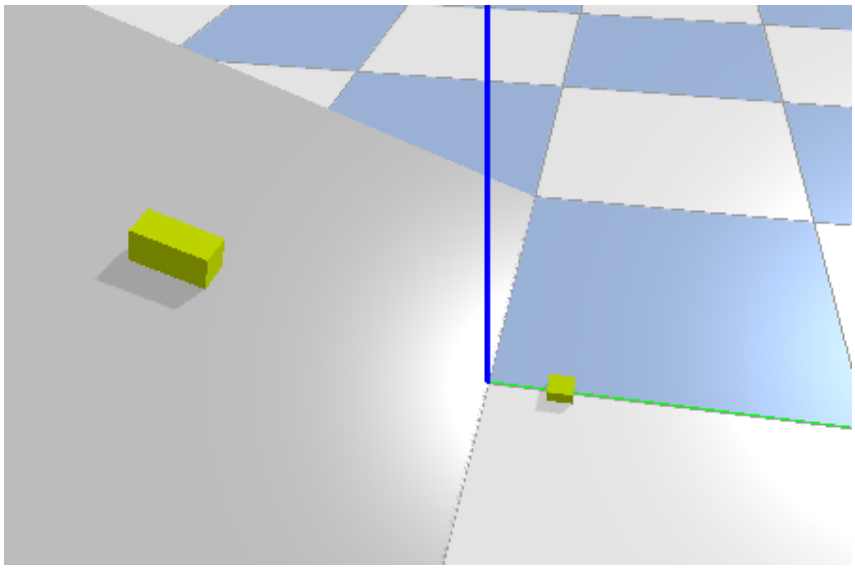


Figure 5.1: Simulation snapshot during dataset generation.

Let us note that this generation process can be formalised using the inverse of f, f^{-1}. For parameter sets which incur movement and collision, f^{-1} is clearly injective. Although the surjectivity of this function is not trivial to explore, the dataset generator assumes that f^{-1}, and implicitly f, are bijective. Additionally, the way data is generated eliminates the need for a dedicated testing set. At test time, new samples can be freely created

and fed to a model. This method ensures that test data is identically and independently distributed from the same space, a vital principle in machine learning.

5.3 Design

A remarkable solution in the area of feature extraction is represented by Convolutional Neural Networks (CNNs). Used mainly for image data [5] [37], their particularity lies in the convolution operation, which excels at identifying local features, but sequentially repeating this process results in a hierarchy of characteristics, of increasing complexities. This relates very well to our problem, as it is different patterns in the velocity profile that can ultimately provide the key information for parameter estimation.

Although it is a design step, identifying an appropriate model architecture inevitably requires analysing the reaction of the model to being trained on a specific dataset. Therefore, we note that the architecture that will be presented is the result of a multitude of tests and experiments with various hyperparameters, such as number of kernels, kernel sizes or pooling layer types.

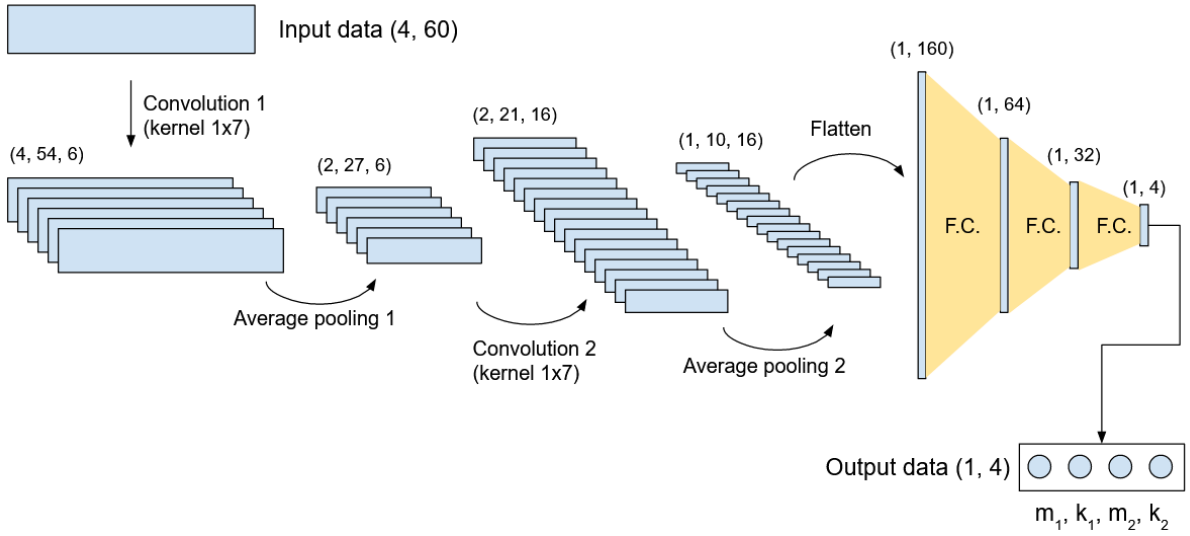


Figure 5.2: Neural network architecture

The input velocity profiles are convolved and then average-pooled two times. Then, three fully-connected layers interpret the extracted features and output a prediction for the four parameters.

5.3.1 Model Architecture

Following a basic CNN architecture, two main phases are involved: feature extraction and feature analysis. The model begins its interaction with the data in a convolutional fashion, to extract the first batch of low-level features. 6 kernels of size (1, 7) are used, and the depth therefore increases to 6. After an average-pooling operation, which halves

the 2D dimensions of the data, another convolutional layer is applied, this time with 16 kernels, further reshaping the data. Again, the results are fed into an average-pooling layer. At this point, the data has dimensions (1, 10, 16), and the feature extraction phase is finished.

Next, these intermediate features are to be used by a set of fully connected layers, in order to ultimately output the four values of interest. Three layers of this type will be used, with 64, 32 and 4 neural units respectively. The latter also represents the output layer. A key decision here is the activation function of these layers – for simplicity, the intermediate ones use ReLU, while the output layer has a linear activation function. An alternative is to use the Sigmoid function, but this would artificially constrain the outputs to the (0, 1) interval, which is, in fact, something that the model should learn on its own.

The complete architecture is depicted in figure 5.2. This is a relatively shallow model, yet appropriate considering the small dimensions of the input data, and the nature of the problem that it aims to solve.

5.4 Implementation

We will now approach the task of transferring this abstract architecture into a computational structure which can be trained and tested. Tensorflow [23] is a Python library developed specifically for this purpose. Enhanced with the Keras API [11], it provides straightforward functionality to describe a NN model as a series of layers. We would like to highlight the `Sequential` model class, upon which our model was developed, as it can be observed in figure 5.3.

```
model = keras.Sequential()

model.add(layers.Conv2D(filters=6, kernel_size=(1, 7), activation='relu', input_shape=(4, 60, 1)))
model.add(layers.AveragePooling2D())

model.add(layers.Conv2D(filters=16, kernel_size=(1, 7), activation='relu'))
model.add(layers.AveragePooling2D())

model.add(layers.Flatten())

model.add(layers.Dense(units=64, activation='relu'))
model.add(layers.Dense(units=32, activation='relu'))
model.add(layers.Dense(units=4, activation = None))
```

Figure 5.3: Code snippet describing the creation of a model using Tensorflow and Keras.

However, this description only instantiates a model, with its complete internal connections, in memory. Overall, this results in 13,252 parameters to be trained, out of which 12,516 belong to the fully connected layers. Additionally, a custom training and testing environment was implemented, to enable repeatedly training and running experiments, so that the behaviour of the model is thoroughly analysed. Each operation is logged, so that changes can be appropriately tracked. We have defined three main branches which trigger different actions, and they will be detailed below.

Training from scratch

The first branch is potentially the most important, as it refers to training the model. The model architecture is fetched, and the compilation operation occurs. It is at this stage that Tensorflow allows the user to define vital aspects related to the training process:

- Loss function – as the model deals with output values less than 1, the `MeanAbsoluteError()` function was used, from the `keras.losses` module. Its alternative, `MeanSquaredError()`, would unnecessarily reduce the amplitude of the error, implicitly reducing its effect on the model weights.
- Optimizer – this refers to the optimisation technique to use during training; our experiments involved ADAM [20] and simple Stochastic Gradient Descent, but no significant accuracy difference was noticed between them, so the final model used ADAM.
- Learning rate – another key hyperparameter, this controls how fast the model moves through the search space. We began by training with a learning rate of 0.001 which was repeatedly decreased by factors of 10 once the loss plateaued.

The next step consists of actually fitting the model to the training data. Here, data is processed in batches over a predefined number of epochs. In each epoch, all training examples perform a forward pass through the model, a cost value is computed using the loss function, and the weights of the model are updated accordingly, using the principles of backpropagation. A custom data pipeline was designed, such that entries are correctly fed into the model from the source velocity profiles dataset. At training time, the dataset consisted of 10,000 entries, generated with the method explained in section 5.2. Experiments showed negligible performance gains when using a considerably larger dataset (e.g. 100,000). Using a batch size of 64 entries, each epoch will have $\lfloor 10,000/64 \rfloor = 156$ steps.

At the end of the training stage, the model is saved to disk in the form of a `.h5` file.

Resuming training

The second branch is very similar to the first one, but it refers to the specific process of continuing the training on a given model. The `.h5` file saved earlier is loaded, effectively restoring the architecture and weights of the model. The model can then be recompiled, to adjust training settings, after which a new training session can begin.

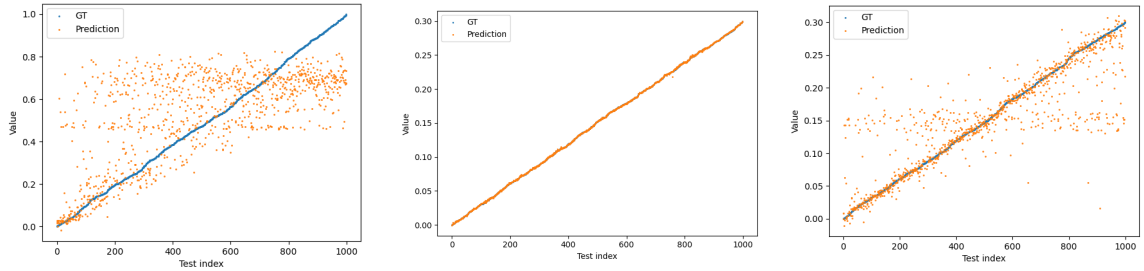
Testing

The third branch disregards the training process and focuses on the experimental side of the problem. The model object is passed as an argument to different methods which can test its performance, as will be detailed in section 5.5.

5.5 Evaluation

The goal of this phase is not only to assess how well the model predicts the four parameters of interest, but also to test whether the model learns the intrinsic relationships between different physical properties.

Generating a test sample is equivalent to generating a training entry: for a set of parameters, we simulate the corresponding scene and extract the velocity profile. This is then fed into the model as input data, resulting in a prediction $\hat{Y}$. The ground truth output data Y is the initial set of parameters, so information about the performance of the model is obtained by comparing Y and $\hat{Y}$.



(a) Values of m_1. (b) Values of k_1. (c) Values of k_2.

Figure 5.4: Comparison between ground truth and predicted values for object parameters 1,000 tests with randomly-sampled parameters were run. (ground truth, prediction) tuples are sorted in ascending order of ground truth value.

Parameter	Error Mean	Error Standard Deviation
m_1	0.1686	0.1412
k_1	0.0004	0.0004
m_2	0.1605	0.1397
k_2	0.0232	0.0375

Table 5.1: Error figures for the four parameters

One of the first tests looked at determining the model's accuracy on the four parameters. Obviously, it is very hard to appropriately explore the entire search space, but random sampling provides a good approximation. Using the technique described above, we sample 1000 tuples for (m_1, k_1, m_2, k_2), and use the model to obtain their corresponding predictions. The error for the four components is then analysed independently. Figure 5.4 shows how the predictions lie, with respect to the ground truth values, for m_1, k_1 and k_2. m_2 had a similar pattern to m_1 and is not depicted for conciseness. Table 5.1 presents the mean and standard deviation of the absolute errors in the scatter plot. Unsurprisingly, the model performs excellently at predicting the value of k_1. For k_2, the error is slightly

higher but still within a reasonable range. For m_1 and m_2, however, the model is still struggling. The following experiments aimed to explain this phenomenon in more detail.

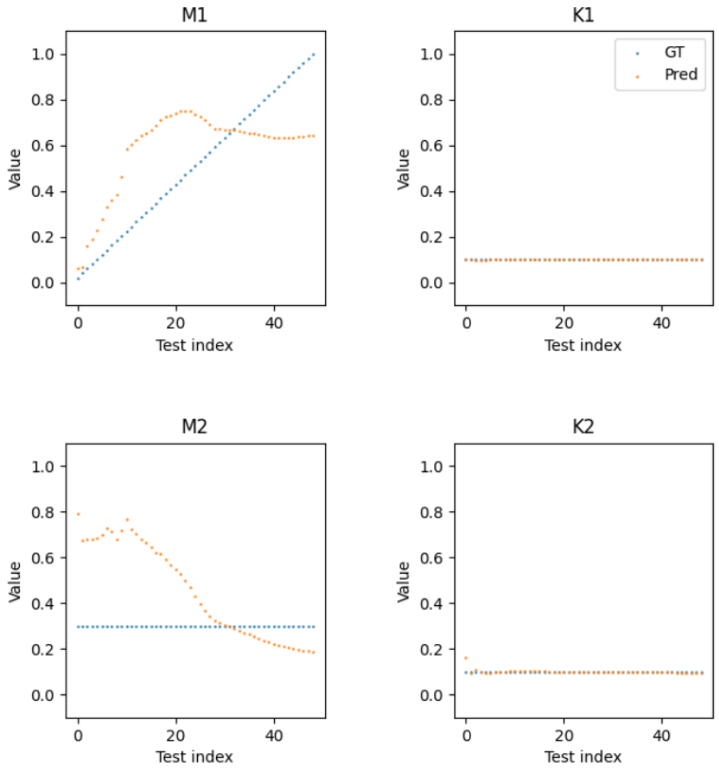


Figure 5.5: Analysing the Neural Network output
All parameters, except for m_1, are constant. Ground truth values are in blue, while orange represents model predictions.

To achieve this, we designed a custom test scenario in which the test cases follow a simple pattern: all parameters are constant, except for one of them, which changes linearly. Results for performing this analysis on parameter m_1 are available in figure 5.5. As it can be observed, the model is far from learning the linear pattern, and subsequent training operations were proven to not ameliorate this situation. This highlighted the point that perhaps the model learns some other relationship between the masses of the two objects. Linking back to section 3.1.1, a key equation defining the collision event is $\phi(m_1, m_2) = \frac{m_1 - m_2}{m_1 + m_2}$. Plotting this function alongside the other values represented an Eureka moment, as it uncovered that the model is indeed able to identify this representative underlying feature in the velocity profile. To further verify this, we plot ϕ and its predicted values using the random sampling method described earlier, to generate the graph in figure 5.6. Therefore, the explicit values for the masses of the two objects are, to an extent, irrelevant.

5.5.1 Assessment

This result uncovered several problems in the dataset and model architecture. First, experiments showed that if $\phi(a, b) = \phi(c, d)$, the velocity profile obtained by simulating with

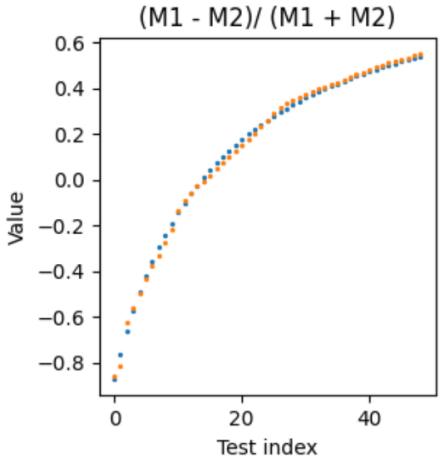
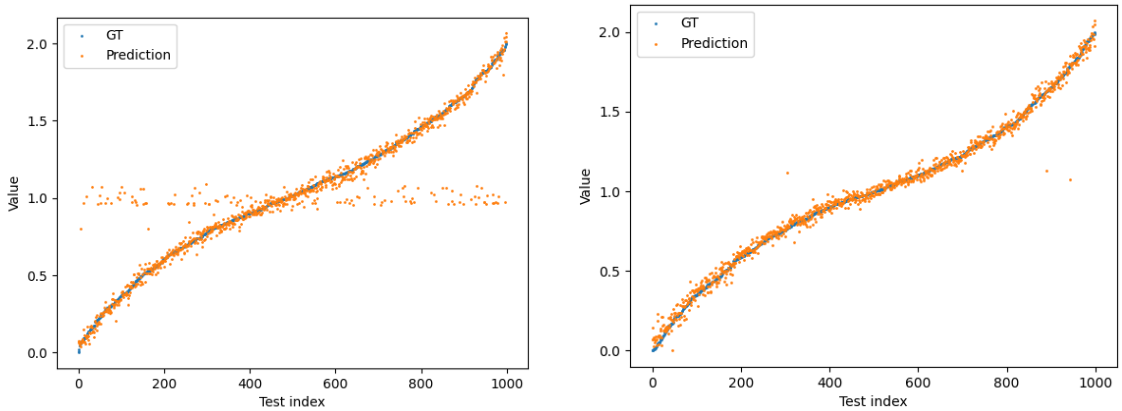


Figure 5.6: Neural Network output when predicting the value of ϕ $\phi(m_1, m_2) = \frac{m_1 - m_2}{m_1 + m_2}$. Ground truth values are in blue, while orange represents model predictions. The model learns this function, which underpins the physical collision event.

parameters (a, k_1, b, k_2) and (c, k_1, d, k_2) will be identical. This contradicts the bijectivity of the mapping function that was asserted during dataset creation and will prevent the model from learning properly, as two identical velocity profiles will potentially be matched to distinct output values. A bijective dataset will remove this limitation and allow correct predictions for the mass values.

A second issue is highlighted by the predictions clustered along a horizontal line at $y = 1$ in figure 5.7a. It was determined that this is caused by the input data size. A velocity profile of length 60 will, for some values of k_1, not capture the full interaction between two objects, as Object 1 slides too slowly. Generating a new dataset with input entries of size (4, 110) and modifying the model accordingly were shown to alleviate this problem, as showed in figure 5.7b.



(a) Results on model which analyses a velocity profile of length 60.

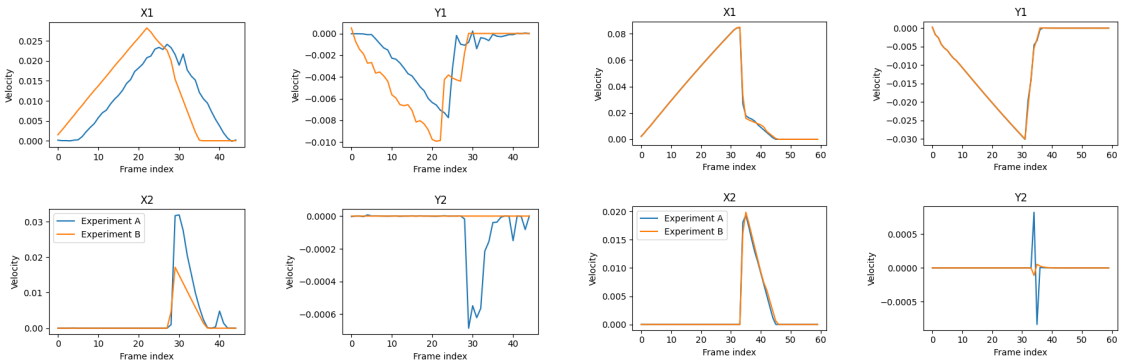
(b) Results on model which analyses a velocity profile of length 110.

Figure 5.7: Values of the ϕ function, using random parameter tuple sampling

Overall, the model successfully fulfils its mission of analysing a velocity profile, identifying key features and extracting the values of the physical properties which determine a certain behaviour. The mass values are not explicitly learned, but this is explained and supported by the underlying movement equations. We would also like to note that this performance is not guaranteed in a scenario where more complex interactions occur. In that case, a higher amount of training data, or a more advanced architecture would be required, to correctly model the phenomena. Symbolic regression [27], for example, provides an appropriate analytical approach which could be used to infer the underlying relationships between the parameters as well as their explicit values. Another method worth highlighting here is OnsagerNet [50], a system which also learns interpretable dynamical models with the help of a neural network, placing higher emphasis on the physical details of the process being analysed. In the current context, however, our proposed network accomplishes this goal with a simpler architecture.

5.5.2 Comparison with the Bayesian model

The performance of the Bayesian model is harder to analyse formally, using concrete error values, as no ground truth data for the physical parameters is available. The two models can nonetheless be compared by exploring their common element: the velocity profile. In the Bayesian case, we compare the video velocity profile with the one yielded by the simulation, after the chains are complete. For the neural network, we compare the input data with the velocity profile generated by simulating with the predicted parameters. As it can be observed in figure 5.8, the neural model performs significantly better than its probabilistic counterpart.



(a) Bayesian model - the observed profile comes from video

(b) Neural network model - the observed profile comes from a simulation

Figure 5.8: Comparison between observed and predicted velocity profiles
In both cases, the predicted profiles are obtained by simulating with the inferred parameters.

However, this apparent advantage comes with several drawbacks. The ML model

depends on a large amount of data, which was in this case trivial to obtain, but might be practically inaccessible in a real-world scenario. Secondly, its performance is limited to the data area explored during training. It is very hard for such models to correctly extrapolate beyond the training domain, and this can be a hurdle if adaptability is a requirement. In comparison, the Bayesian model excels thanks to the versatility implied by its high-level approach, but introduces a trade-off for precision and run-time.

Chapter 6

Project management

6.1 Methodology

We will now describe the process of identifying an appropriate design and development methodology for the applied components of the project.

From the beginning, the goal of the project highlights two opposing natures: firstly, the area that we are exploring is unequivocally broad, yet popular – in-depth studied methods exist, a lot of effort has been put into applying these methods to real-world problems. From this point of view, we have a strong foundation on which we can build. This refers to the scientific models that will be involved (further described in the design section). Secondly, there is a certain uniqueness of the task, determined by both the overarching motivation as well as how the existing resources can be combined in order to reach it. These elements place us in a research-prone position, thus encouraging a less strict software development approach, open to unexpected hurdles or turns (such as Agile). On the other hand, the project is chained to the temporal limitations established in the initial stages, which were in their turn constrained by the university timetable and deadlines. This contrasts the previous point, but emphasises the predetermined character of the work, pushing towards a linear approach (such as Waterfall). We can now shortlist several key factors that we considered at this stage, from the time-work relationship:

- Fixed deadline
- Feasible amount of work per time period (i.e. the available time was sufficient given the expected amount of work to be performed)
- Need to account for potential delays caused by exploration of methods to be employed
- Sensible guidelines available in the reference papers
- Limited compatibility with large-scale coding methodologies (work is individual,

not in a team, goals are not defined by an external stakeholder, deliverables do not constitute a commercial product etc.)

- External factors (interference with other coursework, personal circumstances, etc.)

Given all these elements, a hybrid solution emerges: we can establish a chain-like structure of the project, in which the different stages are linked sequentially, but a buffer time period is reserved. This way, progress follows a linear, predictable trajectory, while the additional time provides the adaptability required for a project of this type. The initial timeline, presented in the project specification, was updated upon completion of the progress report document, and is reproduced in figure 6.1 below. Each of the stages were established in an atypical pre-design stage (with clear specification but just a vague overview of the links between the components), and represent, in fact, a wide timeframe covering all the smaller sub-stages, that were to be uncovered at the time of their specific exploration.

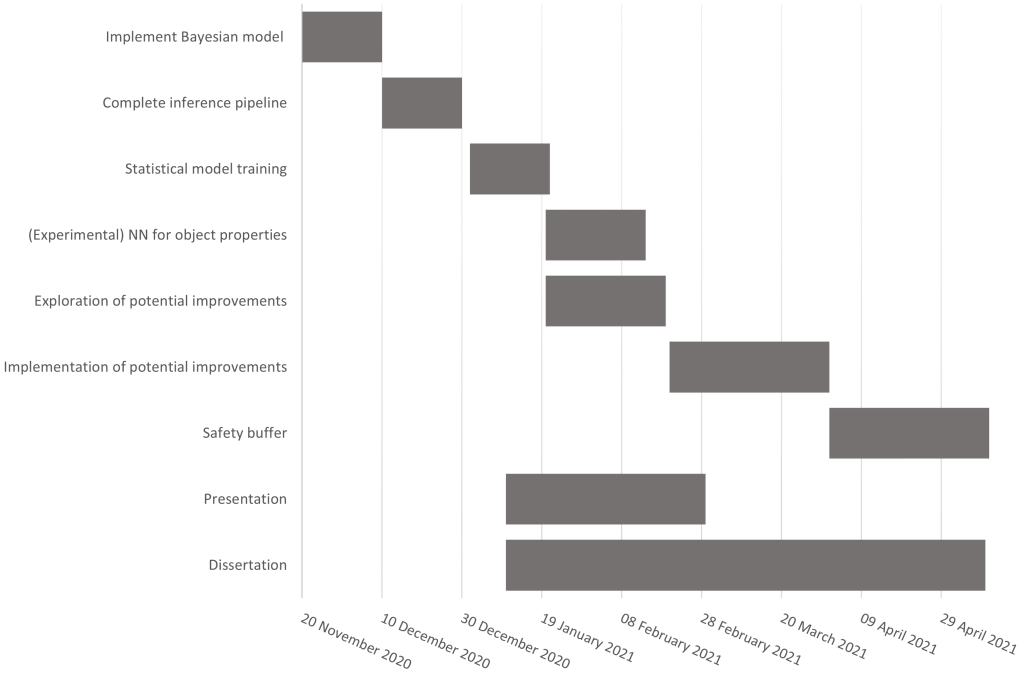


Figure 6.1: Project stages

Methodology assessment

This served as a fixed reference scheme for evaluating progress throughout the evolution of the project. One point that introduced difficulties was the aim to simultaneously develop technical and presentational material such as the presentation slides or the dissertation, as initially planned for the later part of the project. This incurred context switching that affected both processes and is to avoid in further work.

Another issue consisted of incorrect estimation of time required for different stages of the project. The safety buffer was indeed a useful resource in this case, but had a

counterproductive psychological effect, leading to tasks being delayed.

6.2 Legal and ethical issues

A important point to be adressed here is that of the data and libraries being used. The Physics 101 dataset [46] is published as a public dataset, available as a free resource to the scientific community.

The main libraries that are used throughout the project, along with their corresponding license types, are listed in table 6.1. All of these allow for responsible non-commercial, scientific usage, and are therefore suitable to use in this context.

Library	**License Type**
NumPy	3-Clause BSD
OpenCV	3-Clause BSD
PyBullet	MIT
Tensorflow	Apache License 2.0
Keras	MIT/Apache License 2.0

Table 6.1: Libraries used and their license types

The project is not subject to any ethical constraints, as no personal information is processed, stored or accessed, and no human participants are involved.

6.3 Acknowledgements

First and foremost we would like to acknowledge the help and constant involvement of the projects' supervisors. Their guidance was always timely and useful in planning and completing this project.

Secondly, considerable acknowledgement goes towards the work of Jiajun Wu and his team, who developed the Galileo system [47]. They provided an intriguing problem which triggered and motivated our efforts to explore this unique subject.

Lastly, several sources that were used during the implementation stage will be mentioned. The OpenCV library tutorials provided the backbone of the algorithm used for feature tracking using the KLT method [31]. A similar resource was the PyBullet Quickstart Guide [32], which provided the initial steps in understanding the simulation environment. The Tensorflow documentation [41] was also frequently used during the implementation of the neural network architecture.

6.4 Author's Assessment of the Project

The technical contribution of the project lies in the two systems for physical properties estimation using either a probabilistic method or a neural network, applied on velocity profiles extracted from a video. This can have multiple applications, out of which we highlight movement prediction, given the endless possibilities of the physics engine. This contribution is relevant to the subject of Computer Science as it represents an attempt to solve the bigger problem of creating an artificial system which is able to reason about the physical world in a manner, and up to an accuracy level similar to human performance.

This project stands as a research exercise and provides essential information about the behaviour and performance of the described systems. Their limitations come from the highly constrained data/environments they operate on. A real world scenario has the potential to be very unpredictable and hard to represent in a simulation environment, which is a key element in our design. The complexity of such a scenario can also vary greatly - here we are only considering the simple physical system that is the inclined plane.

Chapter 7

Conclusion

Motivated by psychological research in the field of human physical phenomenon understanding, the project asks a rhetorical question – can machines be designed to reason about the physical world in the same way that humans do? Previous work in the field brought to light several attempts to accomplish this, identifying scientific models which could represent the processes happening in the human brain. Specifically, this requires a virtual equivalent of the scene being observed, that the human brain manipulates for various types of inference, and a source of bias or uncertainty, introduced either at observational or analytical level.

A system following these principles was described by Jiajun Wu et al. [47]. We aimed to recreate its architecture and explore limitations and potential improvements. The system analyses videos of objects sliding down an inclined plane and, using a tracking algorithm, extracts the velocity profiles of the objects involved. An equivalent scene is constructed in a physics engine environment, where it can be simulated forward in time, with custom physical parameters for the objects of interest. By comparing the velocity profile extracted from video to the velocity profile extracted from simulation, the system can determine how close its guess for the physical parameters is to the ground truth values – without actually knowing these values. The exploration is guided by a Markov Chain Monte Carlo method, enhanced with a Metropolis-Hastings proposal acceptance algorithm. Once a desired number of chain steps are completed, the system provides a result, in the form of the tuple of object parameters which best approximate the real, observed scenario. We highlight that this information can be applied to the problem of object movement prediction, by simulating the scene forward in time as long as necessary.

Additionally, we experimented with a CNN architecture which, given a velocity profile, is trained to extract the same parameters as the probabilistic model. This can constitute a considerable improvement over the initial method, assuming that the scene being observed can be reconstructed in a simulation world – such that training data can be generated efficiently. Instead of performing the MCMC search to identify optimal parameters, the velocity profile extracted from video can be passed on to the NN, which will analyse it

and produce an accurate prediction. As it was observed, depending on the training data, accuracy does not necessarily refer to explicitly determining one parameter tuple, but to determining a parameter tuple that satisfies the underlying physical equations.

The two models that we developed and analysed propose two essentially different solutions for the same problem. One focuses on the meta-problem of how the result is obtained – following an approach that mimics human reasoning and only secondarily targeting accuracy, while the other can be portrayed as a sturdy solution, having its main focus on the accuracy of the results. Overall, their applications would vary based on specific, real-world requirements, but at this stage none of them is fully adequate for interaction with a wider environment. However, both approaches represent meaningful explorations of the problem field and stand as strong foundation points for systems that aim to replicate either of these behaviours.

7.1 Future work

Our efforts were subject to fixed time and resource constraints, yet an important contribution to the field was made. Nonetheless, many improvement areas have been uncovered and we aim to present some of these in this section.

Perhaps the most important step towards a better performing system consists of enhancing its adaptability. At a high level, this refers to expanding the environment that it can handle, by progressively removing constraints and developing adequate solutions for them. A good place to start is the preprocessing stage involved in the tracking algorithm. This can be replaced by a NN-backed system that interprets the scene and outputs key information that is needed during later processing, such as object shape/size/position or adequate features for tracking.

As our experiments demonstrated, the velocity feature space provides essential but not sufficient information for extracting all parameters of interest - mass, for example, cannot be inferred explicitly. This indicates that, if accurate parameter values are sought, a different feature space is necessary. Deep Learning provides an alternative for this, but, as highlighted in previous works, such a latent space usually disregards the physical meaning of the data and introduces an uninterpretable, abstract component.

This links very well to the idea of analysing alternative methods which aim to determine the intrinsic model which underlies an observed event, such as Symbolic Regression or the OnsagerNet. Although our models - especially the Neural Network - perform remarkably well on the inclined plane scenario, it is inevitable that their limitations will surface once a more invovled scene is used. For example, when increasing the number of free objects, the Bayesian model will have to explore a space whose size increases exponentially, drastically affecting its performance.

At a very low level, future work must include a code refactoring process, which would

reorganise the functionality into adequate classes, following Object Oriented Programming principles.

Bibliography

[1] Pulkit Agrawal, Ashvin Nair, Pieter Abbeel, Jitendra Malik, and Sergey Levine. "Learning to poke by poking: experiential learning of intuitive physics". In: *NIPS'16 Proceedings of the 30th International Conference on Neural Information Processing Systems*. Vol. 29. 2016, pp. 5092–5100.

[2] W.B. Anderson. "Physics for Technical Students: Mechanics and heat. 1st ed". In: Physics for Technical Students. McGraw-Hill book Company, 1914, p. 112. URL: https://books.google.ro/books?id=Pa0IAAAAIAAJ.

[3] Peter Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, and Koray kavukcuoglu. "Interaction networks for learning about objects, relations and physics". In: *NIPS'16 Proceedings of the 30th International Conference on Neural Information Processing Systems*. 2016, pp. 4509–4517.

[4] Peter W. Battaglia, Jessica B. Hamrick, and Joshua B. Tenenbaum. "Simulation as an engine of physical scene understanding". In: *Proceedings of the National Academy of Sciences of the United States of America* 110.45 (2013), pp. 18327–18332.

[5] Ashwin Bhandare, Maithili Bhide, Pranav Gokhale, and Rohan Chandavarkar. "Applications of Convolutional Neural Networks". In: *International Journal of Computer Science and Information Technologies® (IJCSIT®)* 7.5 (2016), pp. 2206–2215.

[6] Kiran S. Bhat, Steven M. Seitz, and Jovan Popovic. "Computing the Physical Parameters of Rigid-Body Motion from Video". In: *ECCV '02 Proceedings of the 7th European Conference on Computer Vision-Part I*. 2002, pp. 551–565.

[7] Wan Botao, Jing Guoxi, Jiang Ying, and Zhang Dianli. *Intelligent service robot for hotel and ward*. 2014.

[8] C.B. Boyer and U.C. Merzbach. "A History of Mathematics". In: Wiley, 2011, p. 232. ISBN: 9780470630563. URL: https://books.google.ro/books?id=BokVHiuIk9UC.

[9] Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. "Discovering governing equations from data by sparse identification of nonlinear dynamical systems". In: *Proceedings of the National Academy of Sciences of the United States of America* 113.15 (2016), pp. 3932–3937.

[10] Thomas Burke. *Construction of the Giza Pyramids*. 2005. URL: https://old.world-mysteries.com/gw_tb_gp.htm (visited on 04/18/2021).

[11] François Chollet et al. *Keras*. https://keras.io. 2015.

[12] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation. Stichting Blender Foundation, Amsterdam, 2018. URL: http://www.blender.org.

[13] Erwin Coumans. *Bullet 2.83 Physics SDK Manual*. 2015. URL: https://bulletphysics.org/ (visited on 04/19/2021).

[14] Erwin Coumans. "Bullet physics simulation". In: *ACM SIGGRAPH 2015 Courses on*. 2015, p. 7.

[15] Flavio Coutinho. *Object Files*. 2019. URL: http://fegemo.github.io/cefet-cg/attachments/obj-spec.pdf (visited on 04/19/2021).

[16] Kenneth James Williams Craik. *The nature of explanation*. 1943.

[17] Jodi Forlizzi and Carl DiSalvo. "Service robots in the domestic environment: a study of the roomba vacuum in the home". In: *Proceedings of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction*. 2006, pp. 258–265.

[18] Clement Godard, Oisin Mac Aodha, and Gabriel J. Brostow. "Unsupervised Monocular Depth Estimation with Left-Right Consistency". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 6602–6611.

[19] I.S. Gujral. "Engineering Mechanics". In: Laxmi Publications, 2008, p. 382. ISBN: 9788131802953. URL: https://books.google.ro/books?id=JM0OG-XUyu0C.

[20] Diederik P. Kingma and Jimmy Lei Ba. "Adam: A Method for Stochastic Optimization". In: *ICLR 2015 : International Conference on Learning Representations 2015*. 2015.

[21] Ziwei Liu, Raymond A. Yeh, Xiaoou Tang, Yiming Liu, and Aseem Agarwala. "Video Frame Synthesis Using Deep Voxel Flow". In: *2017 IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 4473–4481.

[22] Bruce D. Lucas and Takeo Kanade. "An iterative image registration technique with an application to stereo vision". In: *IJCAI'81 Proceedings of the 7th international joint conference on Artificial intelligence - Volume 2*. 1981, pp. 674–679.

[23] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster,

Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/`.

[24] Michael Mccloskey, Alfonso Caramazza, and Bert Green. "Curvilinear Motion in the Absence of External Forces: Naïve Beliefs About the Motion of Objects". In: *Science* 210.4474 (1980), pp. 1139–1141.

[25] Albert Michotte. *The perception of causality*. 1963.

[26] Microsoft. *Windows Subsystem for Linux Documentation*. 2021. URL: `https://docs.microsoft.com/en-us/windows/wsl/` (visited on 04/19/2021).

[27] W Minnebo and S Stijven. "Empowering knowledge computing with variable selection". PhD thesis. Dept. Comput. Sci. Math., Univ. at Antwerp, Antwerp, 2011, p. 48.

[28] Roozbeh Mottaghi, Hessam Bagherinezhad, Mohammad Rastegari, and Ali Farhadi. "Newtonian Image Understanding: Unfolding the Dynamics of Objects in Static Images". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 3521–3529.

[29] Roozbeh Mottaghi, Mohammad Rastegari, Abhinav Gupta, and Ali Farhadi. ""What Happens If..." Learning to Predict the Effect of Forces in Images". In: *European Conference on Computer Vision*. 2016, pp. 269–285.

[30] OpenCV. *Open Source Computer Vision Library*. 2015.

[31] *Optical Flow - OpenCV Python Tutorials*. 2021. URL: `https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_video/py_lucas_kanade/py_lucas_kanade.html#lucas-kanade-optical-flow-in-opencv` (visited on 11/15/2020).

[32] *PyBullet Quickstart Guide*. 2021. URL: `https://docs.google.com/document/d/10sXEhzFRSnvFcl3XxNGhnD4N2SedqwdAvK3dsihxVUA/edit` (visited on 11/01/2020).

[33] *Python package installer*. 2021. URL: `https://pip.pypa.io/en/stable/` (visited on 04/19/2021).

[34] Adam N. Sanborn, Vikash K. Mansinghka, and Thomas L. Griffiths. "Reconciling intuitive physics and Newtonian mechanics for colliding objects." In: *Psychological Review* 120.2 (2013), pp. 411–437.

[35] Michael Schmidt and Hod Lipson. "Distilling Free-Form Natural Laws from Experimental Data". In: *Science* 324.5923 (2009), pp. 81–85.

[36] Jianbo Shi and Tomasi. "Good features to track". In: *1994 Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*. 1994, pp. 593–600.

[37] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *ICLR 2015 : International Conference on Learning Representations 2015*. 2015.

[38] Mark G Sobell. *A practical guide to Ubuntu Linux*. Pearson Education, 2015.

[39] Shuran Song, Samuel P. Lichtenberg, and Jianxiong Xiao. "SUN RGB-D: A RGB-D scene understanding benchmark suite". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 567–576.

[40] Unity Technologies. *Unity Game Engine*. 2021. URL: `https://unity.com/` (visited on 04/19/2021).

[41] *Tensorflow API Documentation*. 2021. URL: `https://www.tensorflow.org/api_docs` (visited on 04/01/2021).

[42] C. Tomasi. "Detection and Tracking of Point Features". In: *CMU Tech. Rep* 1 (1991).

[43] Tomer Ullman, Andreas Stuhlmüller, Noah D. Goodman, and Joshua B. Tenenbaum. "Learning physical theories from dynamical scenes". In: *Cognitive Science* 36.36 (2014).

[44] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[45] Stéfan van der Walt, S Chris Colbert, and Gaël Varoquaux. "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science and Engineering* 13.2 (2011), pp. 22–30.

[46] Jiajun Wu, Joseph J. Lim, Hongyi Zhang, Joshua B. Tenenbaum, and William T. Freeman. "Physics 101: Learning Physical Object Properties from Unlabeled Videos." In: *British Machine Vision Conference 2016*. 2016.

[47] Jiajun Wu, Ilker Yildirim, Joseph J. Lim, William T. Freeman, and Joshua B. Tenenbaum. "Galileo: perceiving physical object properties by integrating a physics engine with deep learning". In: *NIPS'15 Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*. Vol. 28. 2015, pp. 127–135.

[48] *XML Robot description format*. 2012. URL: `http://wiki.ros.org/urdf/XML/model` (visited on 04/19/2021).

[49] Tian Ye, Xiaolong Wang, James Davidson, and Abhinav Gupta. "Interpretable Intuitive Physics Model". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 89–105.

[50] Haijun Yu, Xinyuan Tian, Weinan E, and Qianxiao Li. "OnsagerNet: Learning Stable and Interpretable Dynamics using a Generalized Onsager Principle." In: *arXiv preprint arXiv:2009.02327* (2020).