

Bachelor's Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Estimating object properties through robot manipulation - dataset and benchmark

Jiří Hartvich

Supervisor: Mgr. Matěj Hoffmann, Ph.D.
Field of study: Cybernetics and Robotics
May 2022

Acknowledgements

I would like to thank the people who helped me with this work, namely Lukáš Rustler, Jan Kristof Behrens, Andrej Kružliak and Krystian Mikolajczyk. I would also like to extend my thanks to my supervisor Matěj Hoffmann for being a guiding force in this work.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských prací.

V Praze dne 19. května 2022

.....
Jiří Hartvich

Abstract

Object property estimation deals with the measuring of objects and subsequent estimation of their properties from these measurements. These measurements are performed using sensors such as RGB cameras, depth cameras, robotic manipulators and grippers accompanied by sensors for quantities such as force, torque, pressure, etc. These elements are combined to form a setup, which is then used in tandem with custom algorithms to measure and estimate the properties of physical objects. This work attempts to create a bridge between different physical setups through an open database as well as a benchmark to compare distinct property estimation methods. This work achieves that through an experiment recording and uploading module that uploads recorded experiments to an open Django database using the REST API. This differs from previous works in that it enables datasets created with different tools to coexist in the same overarching data structure. Other works focus mainly on either generating measurement data, property estimations or higher order knowledge, all the while working independently to each other. Using the resulting database from this work it is possible to integrate already existing results from other works into a shared, accessible format. This work is part of the wider IPALM (Interactive Perception-Action-Learning for Modelling Objects) project and contributes an expandable database for robotic manipulation, including grasping, physical objects and benchmarking of property estimation methods.

Keywords: Object property estimation, grasping, robot manipulation, interactive perception

Supervisor: Mgr. Matěj Hoffmann, Ph.D.

Abstrakt

Odhadování fyzikálních vlastností objektů spočívá v měření objektů a následném odhadování jejich vlastností na základě tohoto měření. Jednotlivá měření se provádějí pomocí senzorů, jako jsou RGB kamery, hloubkové kamery, robotické manipulátory a chapadla doplněná o čidla veličin jako síla, moment síly, tlak apod. Tyto prvky pak společně utvářejí sestavu, která se v kombinaci s vlastními algoritmy používá k měření a odhadování vlastností fyzických objektů. Cílem této práce je vytvořit jednak most mezi různými fyzikálními sestavami v podobě otevřené databáze a jednak systém pro porovnávání různých metod odhadování fyzikálních vlastností. Toho se práce snaží dosáhnout pomocí modulu, který zaznamenává experimenty a následně je za využití REST API nahrává do otevřené databáze, vytvořené pomocí knihovny Django. Na rozdíl od předchozích prací tato databáze umožňuje různým datasetům vytvořeným odlišnými způsoby koexistovat ve stejné, nadřazené struktuře. Jiné práce se zabývají především buď generováním dat, odhadováním fyzikálních vlastností, nebo odvozováním znalostí a obvykle vznikají nezávisle na sobě. Díky databázi vytvořené v rámci této práce je možné začlenit výsledky z jiných prací do sdíleného, otevřeného formátu. Práce je součástí širšího projektu IPALM (Interactive Perception-Action-Learning for Modelling Objects), do něž přispívá rozšiřitelnou databází pro robotickou manipulaci, zahrnující uchopování objektů, popisování fyzických objektů a porovnávání algoritmů na odhadování vlastností.

Klíčová slova: Odhadování vlastností objektů, uchopování objektů, robotická manipulace, interaktivní vnímání

Překlad názvu: Odhadování fyzikálních vlastností objektů pomocí robotické manipulace – dataset a srovnávací měřítko

Contents

1 Introduction	1		
2 Related Work	3		
2.1 YCB (Yale-CMU-Berkeley Object and Model set)	3		
2.2 GRASPA (Robot Arm graSping Performance benchmArk)	3		
2.3 EGAD (Evolved Grasping Analysis Dataset)	4		
2.4 RoCS (Robot-Centric dataSet) . .	4		
2.5 ROS household objects	4		
2.6 BURG (Benchmarks for Understanding Robotic Grasping) . .	5		
2.7 Individual property datasets	5		
2.7.1 Single-grasp deformable object discrimination	5		
2.7.2 Elasticity estimation of soft objects using robot grippers	5		
2.7.3 YCB-impact sounds dataset . .	5		
2.8 Summary	5		
2.9 Relation to current work	6		
3 Materials and Methods	9		
3.1 Experimental Setup	9		
3.1.1 Physical Setup	9		
3.1.2 Measurement Pipeline	10		
3.1.3 Exploratory actions	10		
3.2 Data Format	11		
3.2.1 Top-down View	11		
3.2.2 Bottom-up View	12		
3.3 Django	15		
3.3.1 Django Rest Framework	16		
3.3.2 Overall structure	16		
3.4 Data Storage Pipeline	17		
3.4.1 Butler	17		
3.4.2 Butler implementation	23		
4 Experiments and Results	27		
4.0.1 Database structure	28		
4.0.2 Measurement	28		
4.0.3 Physical measurement	30		
4.0.4 Formatting	31		
4.0.5 Uploading	32		
4.0.6 Database	33		
5 Conclusion, Discussion, and Future Work	39		
Bibliography	41		

Chapter 1

Introduction

In recent years, digital datasets have become commonplace, making it easy to compare the performances of learning-based techniques anywhere in the world. Especially computer vision has received a lot of attention and datasets such as ImageNet [1], CIFAR-100 [2]. Both are easily downloadable and enable us to benchmark computer vision algorithms. The key to the success of many of these techniques is the widespread availability of data and clear benchmarking protocols to compare the algorithms. This enables researchers to iterate on the best ideas to further push performance past previous methods. Images, videos—digital data in general—are easily transportable by internet and easy to produce in great quantity, therefore ideal for benchmarking. In addition, image sensors, i.e. cameras, are ubiquitous and highly standardized instruments, making it easy to produce consistent and applicable data across the globe.

The downside is that cameras provide only a partial snapshot of reality. It is necessary to measure other physical quantities to form a more comprehensive description. To fill in that gap one has to measure the object of study using other modalities, such as touch, sound, etc. Sensors for some these include pressure sensors, force sensors, microphones and depth cameras.

When creating a dataset of a physical property of an object using a sensor other than a camera, we measure objects with the sensor and upload the data to a server on the internet. However, the type of sensor used to measure the property may not be as standardized as cameras are. If we take a picture with two different brands of cameras, the results are almost indistinguishable. Not only is the sensitivity to different wavelengths carefully calibrated but the measurement is also direct, which means that both cameras measure photons actually reflected off that object. On the other hand, stiffness, for instance, can be measured directly by a force sensor or inferred from the electrical current in a gripper actuator. Both methods are theoretically able to provide an accurate description of said property. However, the conversion from sensor outputs to force is generally nonlinear.

Now the issue arises when dealing with physical quantities: they are bound to physical objects. It is possible to obtain something akin to ground truth values by measuring physical objects' properties professionally [3] and then base algorithms on those values. This is unsustainable owing to the fact that objects do not commonly have their properties uniformly distributed—a bottle for example has a body and a neck, the neck being hard and the body being softer. What does it then mean to know the stiffness of a water bottle? And how can a simple water bottle be saved into a digital dataset? Is it possible to have

an easily distributable dataset of physical objects?

A famous example that fulfills these requirements is Willy Wonka's famous television. It transmits chocolate over the air directly to your home television set, from where it can be grabbed and thoroughly enjoyed [4].

The closest we come to this idea from science fiction is in the form of standardized datasets. One such dataset is the YCB object and model dataset. It consists of common and easily graspable standardized physical objects that, upon request, are sent to whoever orders them. This enables different labs to benchmark their own grasping and other algorithms on the same physical objects.

What if the YCB dataset cannot be ordered or if objects not present in the dataset are to be explored by robotic techniques? In that case it is necessary to digitally approximate the objects as closely as possible so that it is possible to share them with others.

That is where the current work comes in. The goal is to develop a system to catalogue measurements and seed a database of common household objects, starting with YCB objects, and to enable others to use the same system to add their own measurements of objects regardless of their belonging to a dataset. Physical objects cannot be stored as values in a database, so they will have to be indirectly represented by a unique ID, images and other measurements. Not only is it desirable to have a system in place of how to store information about these objects, it is also key to have the information easily accessible and, in case of own measurements, easy to upload. Unlike image datasets that are often stored as large zip files never to change again, it is desirable that this database of measurements is expandable, all the while enabling labs without robotic setups of their own to use the uploaded data to benchmark algorithms of their own.

Chapter 2

Related Work

This section introduces related works mainly in robotic manipulation. The main focus of this work is saving measurement, property estimations, grasps and other information about physical objects. Thus, so as not to repeat work other people have already done, we shall concisely introduce works related to our goal and what they aim to achieve, how they compare to each other and how their goals leave space for our work.

2.1 YCB (Yale-CMU-Berkeley Object and Model set)

The YCB object and model set consists of common household objects [5]. The 3D models of these objects are publicly available for download at <https://www.ycbbenchmarks.com/>. The models come with their respective textures gathered from images of the objects on a rotating pedestal as measured by the authors of the dataset. The virtual versions also contain labelled images for image processing algorithms. The physical version object set can be ordered on the YCB dataset's website.

The YCB dataset is designed for facilitating benchmarking in robotic manipulation. The set consists of everyday objects with different shapes, sizes, textures, weight and rigidity. The article proposes several grasping and object manipulation benchmark protocols for robot manipulation research.

A large contribution of this article is the standardized household objects that are easily obtained by simply ordering all objects as a set. The objects are rich in physical features in contrast to homogenous 3D-printed objects, making it a more suited dataset for other applications as well, such as physical property estimation.

2.2 GRASPA (Robot Arm grasping Performance benchmArk)

GRASPA is a benchmark to evaluate how well a particular setup grasps given objects [6]. It calculates a grasping score assuming that the grasping position can be reached by the gripper. The grasping score can be summarized by the following function:

$$f(p, g, o, s) \rightarrow \text{grasping score}$$

where p is the set of reachable positions, g is the set of grasps corresponding to their respective positions in p , o is the objects of interest and s is the setup being evaluated.

The reachability on a given area is determined using inverse kinematics and auto-sensory information and using visual processing from outside the setup. The objects are then placed in reachable positions and testing begins.

Similarly to the previously mentioned YCB, GRASPA also focuses on grasp benchmarks but unlike EGAD in the following section, it only focuses on evaluating grasping algorithms. Some papers propose methods how to grasp given some data; this paper proposes a scoring metric of already generated grasps.

2.3 EGAD (Evolved Grasping Analysis Dataset)

EGAD utilizes evolutionary algorithms to create a set of objects specially designed for grasp generation and benchmarking [7]. To that end, two main metrics for evaluating the complexity and difficulty of grasping were created. The objects are ranked in one dimension by complexity ranging from simple to complex and by difficulty of grasping along the other dimension, all the while maximizing geometric diversity. The dataset provides around 2000 3D models uniformly covering the whole feature space. This is a significant improvement from previous grasping datasets such as YCB or Dex-Net 2.0 [8] in covering all possible grasping configurations. In addition, the authors propose 49 objects representing each combination of the two dimensions in seven levels of of complexity/difficulty. This is to provide an even simpler and more standardized grasping benchmark.

The objects contained in this dataset were originally palm-sized, however due to their digital nature, the size can be adjusted before printing to account for the gripper aperture. EGAD still leaves space open for a deformable object dataset since it only rates the difficulty of the objects under the assumption that everything is rigid.

2.4 RoCS (Robot-Centric dataSet)

This article proposes a method of creating conceptual understanding of objects from the point of view of a robot [9]. The authors combine existing methods of property estimation to acquire physical properties. It accounts for inputs from only the robot itself, hence robot-centric. The authors propose an extensible property estimation framework which consists of methods to obtain first quantitative measurements of physical properties (according to their paper) such as rigidity, hollowness, heaviness, etc. and then functional properties such as containment, support, etc.

2.5 ROS household objects

ROS household objects is a ROS package that is an implementation of a database that catalogues some physical objects [10]. It is the one among the related works that most resembles what this work aims to achieve. It contains some household objects, their 3D models, who the manufacturer is, their commonly used name and some other properties. The issue with this package was that it was not extensible by any user other than the maintainer of the package. It was originally intended to be an online database of objects,

their grasps and other descriptions. Although it is made for the ROS version Indigo from the year 2014, it contains some structure that is of interest. It has since been taken offline but still remains available today as a ROS package.

2.6 BURG (Benchmarks for Understanding Robotic Grasping)

BURG builds on top of the previous dataset created by the Technical University in Vienna, namely ARID and OCID [11]. OCID (Object Clutter Indoor Dataset) combines the datasets ARID and YCB into an expanded RGBD-dataset containing point-wise labeled point-clouds for each object [12]. ARID (Autonomous Robot Indoor Dataset) is a large-scale, multi-view object dataset collected with an RGBD camera mounted on a mobile robot [13],

2.7 Individual property datasets

In the laboratory here at CTU there have also been created a few datasets. Two of them consist of measurements and estimates of physical properties, while the other dataset uses various types of grippers to generate measurement data and uses it to discern different objects made of materials with varying stiffnesses. The grippers used in these works at the lab are: Barrett hand, qb SoftHand, Robotiq 2F-85 and OnRobot RG6 [3].

2.7.1 Single-grasp deformable object discrimination

This is the dataset where grippers are used to discern objects among one another [14]. The crux of this work is examining learning-based techniques that can be adapted to learn to recognize physical objects. In this case most of the objects are visually similar deformable objects but with different stiffnesses.

2.7.2 Elasticity estimation of soft objects using robot grippers

This work is a collection of the output data from the various grippers mentioned in Section 2.7 and objects' elasticity estimations [15]. It also provides some methods to estimate elasticities from the measured data.

2.7.3 YCB-impact sounds dataset

This dataset provides a set of sounds generated by hitting, scratching and dropping objects of the YCB model dataset [16]. Sound is one of many modalities with which physical objects can be measured, and in this work it shall indeed be used as one of several.

2.8 Summary

The data from these datasets can well be used to populate the physical object database in the form of measurements for further processing as well as property estimations. For a

Table 2.1: Comparison of features from the related works.

Dataset	Physical objects	Measurement	Property estimation	Grasp proposals	Benchmarks
YCB	✓	×	×	×	×
GRASPA	×	×	×	×	✓
EGAD	✓	×	×	×	✓
RoCS	×	×	✓	×	×
Household objects	✓	×	×	✓	×
BURG	✓	✓	✓	✓	×
Object discrimination	✓	✓	✓	×	×
Elasticity estimation	✓	✓	✓	×	×
Sounds dataset	✓	✓	✓	×	×

comprehensive comparison of functionalities of the mentioned works see Table 2.1.

2.9 Relation to current work

This work aims to fill in the gaps in robot manipulation left unresolved from the related works. If the goal is to gain information from working with objects at hand, then existing methods are already capable of fulfilling that task. However, obtaining a complete description of an object is exceedingly difficult. The previous works did not attempt to provide a complete description of a few objects, but to encapsulate a certain aspect of the object or interaction with it. For example, GRASPA is a comprehensive benchmarking method for object grasping. YCB provides the physical instances as well as the complete 3D models of a few household objects. EGAD provides comprehensive benchmarking objects over the whole space as defined by the authors.

RoCS ultimately provides a complete set of measurements of a few pre-defined properties on a distinct set objects as well as a high-level framework of knowledge creation on within the context of a *single* setup.

Regarding RoCS, it is closest in that it provides some set of physical properties. In their work, the authors selected flatness, hollowness, size, roughness, rigidity and heaviness as physical properties. Note that these properties are significant only from the conceptual standpoint of understanding the objects, according to the authors.

This differs from the other mentioned works as the main focus here is the understanding of objects from the robot’s perspective. In other words, gaining information about the objects independent of which physical setup is used for measuring.

What the authors in each work set out to do is to propose a closed set of problems and

subsequently attempt to resolve each one of them. Yet, these tasks are related, and there have been attempts at linking the problematics of household object manipulation, grasping, property estimation, etc. The role of the mentioned articles in the context of an information gathering pipeline can be seen in Figure 2.1.

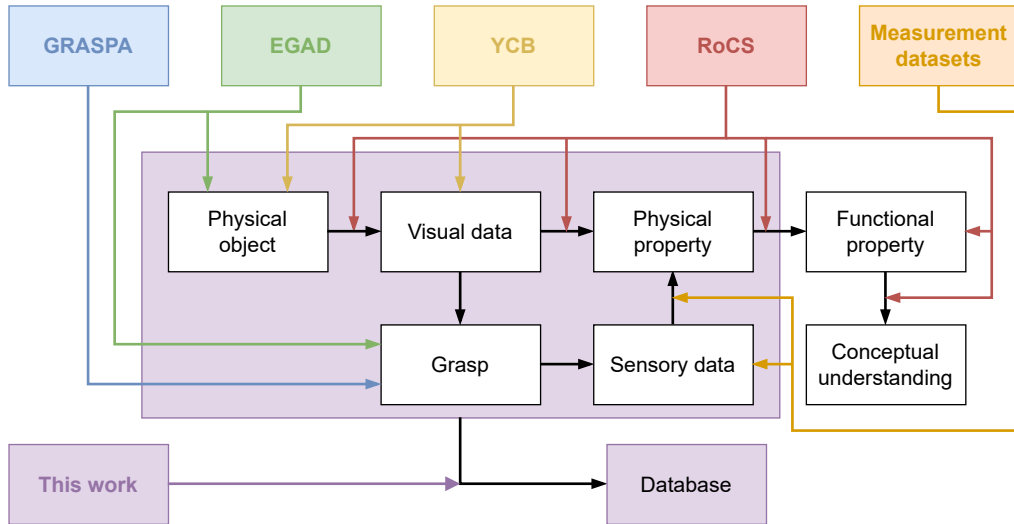


Figure 2.1: The semantic relationship between the GRASPA benchmark, the EGAD grasping dataset, the YCB physical object dataset, and the RoCS framework.

The problem with the aforementioned databases/datasets is that they are not expandable without the authors’ consent and at times setup/version specific.

The relationship between the related works, their contributions and our goal are depicted in Figure 2.2. The lack of a link from each element to an open and online database is represented by crossed out arrows.

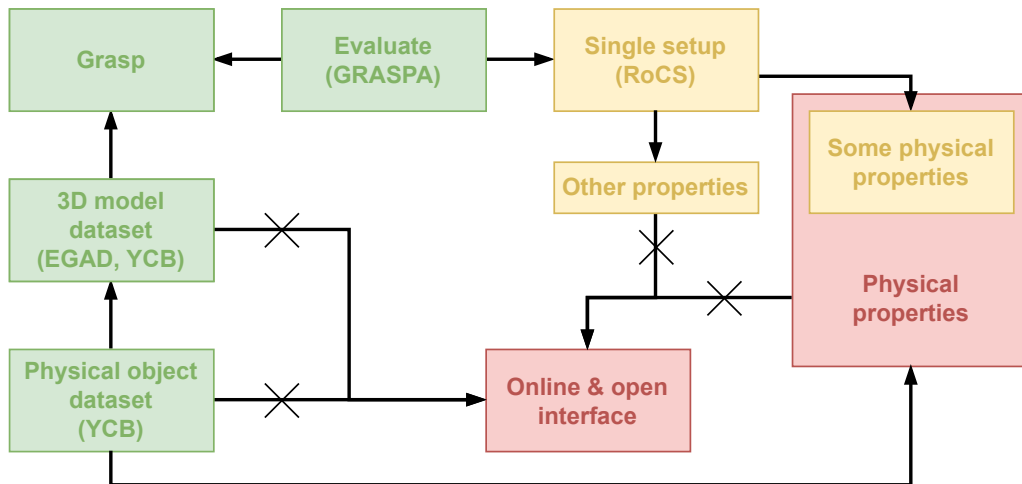


Figure 2.2: Current structure of object and setup relationships.

In conclusion, the database ought to have multiple sources of data, easy uploading, easy

Chapter 3

Materials and Methods

3.1 Experimental Setup

This section deals with the physical setup and the pipeline for obtaining measurement data. The physical setup is the same as in [17] and it will be used to perform experiments to gather data for further processing. The software part of this section will be built on the data gathering pipeline developed in parallel with this work.

3.1.1 Physical Setup

The physical setup consists of two Intel D435 depth cameras, a Kinova Gen3 arm with built-in torque sensors and an attached Robotiq 2F-85 gripper and a RODE VideoMic Pro; see Figure 3.1.



Figure 3.1: The physical setup.

Working with the physical setup is mostly handled by the data generation pipeline. What is necessary to perform manually is object selection and placement. The dataset primarily used to seed the database and test the pipeline is the YCB object dataset and some handpicked in-house objects.

■ 3.1.2 Measurement Pipeline

Three principal components of the measurement pipeline are the following:

- Physical Setup - hardware & objects
- Action selection & property estimation algorithms
- Outputs - measured data & estimated properties

Out of these, the physical setup and outputs are of interest to this work.

■ 3.1.3 Exploratory actions

The measurement actions are controlled by the action selection framework [17] and consist of a selected collection of *actions*. Each action measures some property, e.g. object category. Actions include squeezing for elasticity, vision for category and material classification, weighing for mass measurements and tapping for material classification from sound.

■ Squeezing

The first action we introduce is squeezing. For the gripper used in this work, the Robotiq 2F85, elasticity estimation consists of recording gripper position and current data and extrapolating from that the elasticity coefficient as per [15].

■ Vision action for object category & material

The vision action was extended as part of this work. Originally the vision action only determined the most likely category of the object on the image. Now it outputs a distribution of object category estimates as well as material estimated. The process of adapting the Detectron2 to suit our needs is as follows.

1. Train a MobileNetV3 classifier on the dataset MINC-2500 [18]
2. Input the camera image into a Detectron2 instance trained on the dataset from [19].
3. Take the bounding boxes produced by Detectron2 and feed them into the MobileNetV3 instance.
4. Plug the same bounding boxes back into Detectron2 with a lowered output threshold to get a distribution of object categories and calculate a weighted average with the category output from the first pass through Detectron2 from step 2.

5. Output the categorical probability distributions of categories and materials that each sum up to one respectively.

■ Weighing

The weighing action is performed indirectly using joint torque sensors already built into the Kinova Gen3 robotic arm. First the script weighs the robot’s own arm, then grasps an object and records the data. Given that the length of the arm is known, the mass is then easily calculated using the Equation 3.1

$$m = \frac{T - T_0}{lg}, \quad (3.1)$$

where T is the measured torque, T_0 is the torque with an empty gripper, l is the length of the arm segment from the measuring joint to the gripped object and g is gravitational acceleration [20].

■ Tapping

Another method of material estimation is tapping. This method in particular, mentioned in Section 2.7.3, uses a neural network to analyze a spectrogram of the recorded sequence to estimate the material composition [16].

■ 3.2 Data Format

The goal is to describe physical objects with as high fidelity as possible. Let us examine the definition of an object. According to the Collins English Dictionary “An object is anything that has a fixed shape or form, that you can touch or see, and that is not alive” [21].

Since it is impossible to store an object in a database, the closest digitally representable approximation is its aspects—physical properties. Physical properties are usually denoted in some combination of SI units, such as mass with the unit of kilogram [kg], dimensions with the unit of meters [m] and other derived quantities.

■ 3.2.1 Top-down View

Object properties are obtained by measuring physical object instances. Physical objects cannot be quickly transported over great distances at ease, so there has to be a way to transmit information about them—their properties.

At the lab we have the YCB physical object dataset. We identify these objects by the fact that they belong to the YCB dataset and by their ID—or unique name—in the dataset.

Otherwise, when there were other sets of objects created from a mix of miscellaneous objects and YCB objects, each object had a new unique name assigned to it. For example there was a real banana and a plastic banana from the YCB dataset used. The representation

of the plastic banana from YCB in the new set is `banana_ycb`. If we combine the knowledge about the dataset name, the ID the banana has in the dataset and the name that was used in the object set from the lab, we get the following representation:

- Plastic banana from YCB:
 - Dataset: YCB
 - ID in dataset: 011_banana
 - Unique name: banana_ycb

The unique name may be unique in the context of the object set in the lab, but it is not unique in general. Let us then call it the “common name” of the object. For the banana we know that it is a banana and that it comes from the YCB object dataset, hence `banana_ycb`. The banana did come to us from the YCB dataset but it may not originally have been made for that dataset. The banana actually comes from the maker of these plastic bananas. Therefore, the “maker” of an object is another identifier.

In conclusion, there are going to be four predetermined fields that the user may fill out for an object instance:

- Object instance
 - Dataset
 - Identifier in dataset
 - Maker
 - Common name

We will call this `ObjectInstance` in the database. A detailed description will follow after other parts of the database have been described.

■ 3.2.2 Bottom-up View

Let us now examine the object instances from the ground up. We have established that a combination of physical properties is a valid way to represent real instances of objects. Among some common quantities that can be measured are elasticity, mass, object category, material category, box size and others. Let us then start from these. We shall go through each property and see how each one can be concisely and comprehensively represented in a data structure.

■ Measurement representation

Before we examine specific properties, let us establish the representation of a measurement. Due to intrinsic errors in measuring instruments, outside factors, low precision or accuracy in the data processing algorithms and other unaccounted-for uncertainties, the measurements

and property estimations are expected to come with an uncertainty attached. Furthermore, the distribution of uncertainties is, for simplicity's sake and as explained in [17], assumed to follow a normal Gaussian distribution, see Equation 3.2.

$$X \sim \mathcal{N}(\mu, \sigma^2). \quad (3.2)$$

■ Types of properties

There will be two types of properties used in this work:

- Continuous property: It has a *name*, a *mean value*, a *standard deviation* and *units*.
- Categorical property: It contains *name* : *value* mappings, where *value* $\in [0, 1]$ and $\sum_{i=1}^n value_i = 1$.

■ Elasticity

It is simply a continuous property, which means it has a value, standard deviation, units and name. Elasticity can then be represented like so:

- Continuous property: elasticity
 - Name: elasticity
 - Mean: μ
 - Standard deviation: σ
 - Units: [Pa]

For the property's visualization see Figure 3.2.

■ Mass

Mass can be represented the same way as elasticity, leaving the results unchanged.

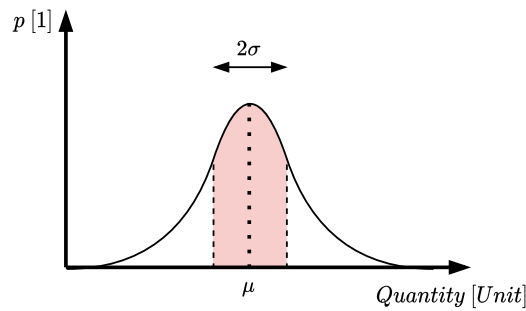


Figure 3.2: Single continuous property.

■ Object/material category

Object category and material category are different from the previous properties in that they are categorical. Thus, the distribution of the property's values is represented by a discrete distribution of points, each with an associated probability p_i , where i is the category; it follows Equation 3.3. See Figure 3.3 for a visual representation of a categorical property.

$$\sum_{i=1}^n p_i = 1 \quad (3.3)$$

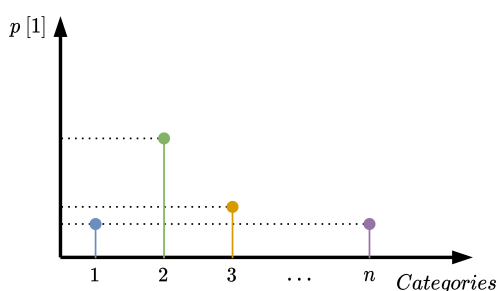


Figure 3.3: Categorical property distribution.

A categorical property can be represented by a vector of [property name, probability, category]. An example of this structure is depicted in Figure 3.4. The numbers add up to one. The property name field is usually the same for all elements, therefore it is sufficient to have a property-wide name for an instance of a property measurement.

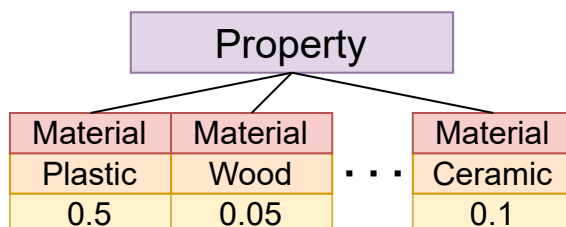


Figure 3.4: Categorical property structure.

■ Size

The next property that is relevant to describing an object is its 3D box size—the x, y, z dimensions. If a user wants to enter a bounding box measurement for the three dimensions x, y, z, it might seem like a categorical property. This is a fallacy as each dimension separately is actually a continuous property like elasticity or mass. Box size can thus be represented as a vector of continuous distributions; see Figure 3.5.

Other properties might have even more dimensions, so the database shall have a general structure to accommodate variable length vectors of property entries, represented in Figure 3.6.

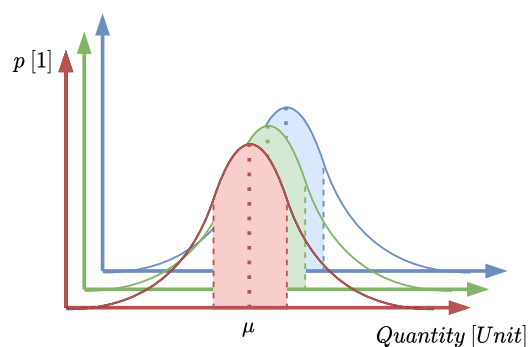


Figure 3.5: Generalized continuous property data structure.

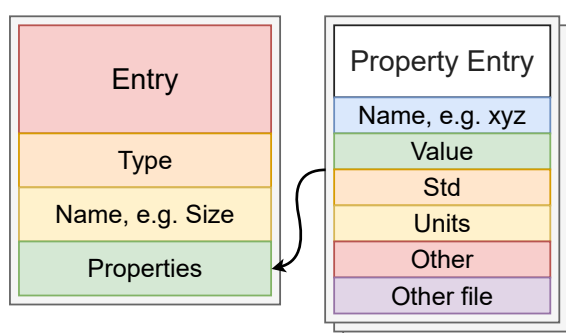


Figure 3.6: Entry structure compatible with the both categorical and continuous properties.

In addition, one might want to save a 3D mesh of the object or other types of properties. To accommodate that, the fields `other` and `other_file` can be used. We can now have a unified approach for storing object properties. The representation of a few properties with this approach is depicted in Figure 3.7.

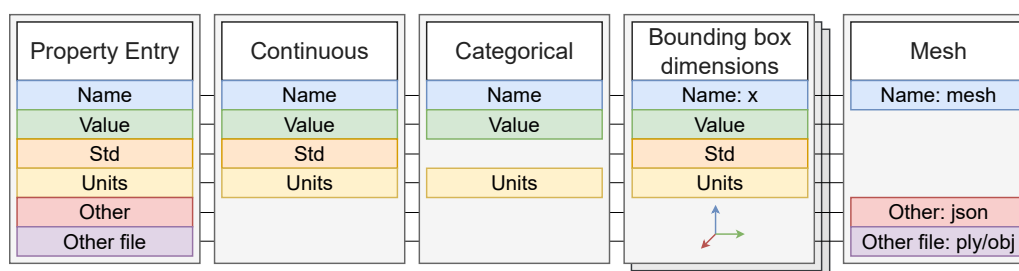


Figure 3.7: Generalized property entry structure and type examples.

■ 3.3 Django

Since Python is easy to use, it shall also be used to implement the back-end for the database. There are several back-end libraries to choose from, each with their pros and cons. Three of the most well-known libraries for server back-end are Flask, Django and FastAPI.

- Flask

- Light-weight
- Little structure
- Django
 - More overhead
 - More structure & sustainable development
- FastAPI
 - Light-weight
 - High-performance

Given some prior experience in Django development and the fact that the task at hand is most closely related to data structuring, it is reasonable to choose Django for this application. It has built-in SQLite3 and PostgreSQL support and the SQL handling is done in the Django library [22]. Django uses its own Object Relational Mapping (ORM) language to receive queries from users and relay them to the SQL database, making the usage of SQL features seamless in Python [23].

■ 3.3.1 Django Rest Framework

Interfacing with the server is usually done in a web browser by default. However, our application requires a quick turnaround time and a way to communicate with the server directly. This can be achieved through the REST API which is a protocol for communicating with servers directly. A REST-enabled server is sometimes called RESTful [24].

To summarize the REST API, by default it enables the user to selectively access data from the website without parsing any HTML code. The website can then be treated as if it were a JSON dictionary, the only downside being higher latency and lower bandwidth.

We shall furthermore enable users to upload data after authentication. In addition, REST is platform independent and has no prerequisites when it comes to using it. As opposed to the works mentioned in Section 2.9, this is a great merit of REST.

■ 3.3.2 Overall structure

We will be modelling the data representation in Django's Object Relational Mapping (ORM). In it, objects have predefined fields and relationships are represented by one-to-one, many-to-one and many-to-many links.

Let us stitch together the objects we have proposed in previous sections. We have established that a measurement object is necessary to bind the measurement data to an object instance. The rules are the following:

- A physical `ObjectInstance` may have multiple `Measurements`.

- Multiple **Measurements** may be performed on a single **Setup**.
- Multiple **Properties** may be estimated from a single **Measurement**.
 - A **Property** may have more than one **Elements**.
- A **Measurement** is assumed to have up to one **Grasp** associated with it.
- A **Measurement** may contain multiple **SensorOutputs**.
 - A **Setup** is made up of multiple **SetupElements**.
 - A **SetupElement** has multiple associated **SensorOutputs**.

The above described structure is visualized in a graph in Figure 3.8.

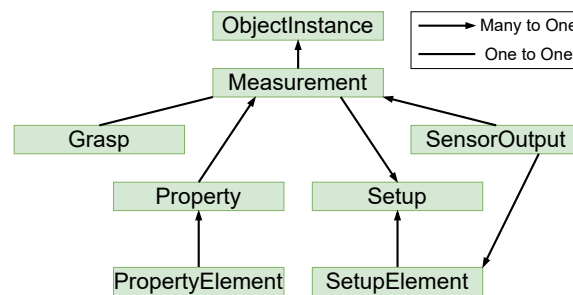


Figure 3.8: Overall structure of an entry of a real object instance.

3.4 Data Storage Pipeline

Since it is impossible to know every setup, the physical setup has to be treated like a black box. That means that data extraction has to be as non-intrusive as possible.

There are several ways to obtain the measured data: either let the user handle the data saving or create a library of functions into which data is entered in a predefined format, then saved. Another method related to the latter option is function decorators, as will be mentioned in Section 3.4.1. The tool developed to help with this task has been called **Butler** as a shorthand.

After this, the data will have to be uploaded. For ease of use, REST API shall be used to interface with the server. A general overview of the steps required to move the data from a local setup to the server is described in Figure 3.9. There are three steps for the data in the chart. The first one is the information after it is output by a particular measurement, the second one is the data stored in memory or on disk which is to then be uploaded to a server, making the data widely available.

3.4.1 Butler

The first step in data extraction is getting the information from local algorithms. The setup and data generation algorithms have to be treated like a black box, but the output data generally follows some rules. The data structure is usually along the lines of Figure 3.8.

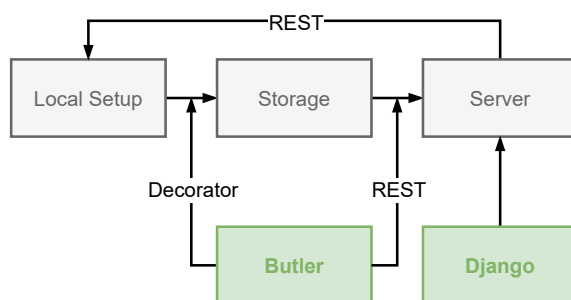


Figure 3.9: Data storage pipeline overview.

■ Measurement Object

The first type of output we shall consider is a so called measurement object, as per the thesis [17]. This will encapsulate both of the aforementioned continuous and categorical property estimations. The object shall be represented by a python class which means that it is possible to save almost any kind of data into it, thus making it possible to save the data entries as demonstrated in Figure 3.7.

The following example is the output format of a continuous property. The `values` field is the sensor outputs—raw data—that are used to estimate the `mean` and `std`—the property estimation—in `params`.

The `std`, or standard deviation, generally stems from several sources such as the sensor, algorithm or the environment. At the lab we obtained the `std` by performing many measurements and extracting the standard deviation from that data. For that reason `std` is assumed to accompany a measurement.

For a categorical property estimation the only difference is that the `params` attribute would be a set of key-value pairs with the keys being the category names, the values being the probabilities of that particular category, all the while the sum of the values being one.

```

class MeasurementObject:
    meas_prop = "elasticity"
    meas_type = "continuous"
    params = {"mean": 100, "std": 10}
    # params = {"cat1": 0.2, "cat2": 0.3, "cat3": 0.5}
    values = [100, 110, 90, ...]
    meas_ID = 0
  
```

Now if one wishes to save the measurements, it is best to save them to disk lest they be lost. For ease of debugging, visualization and general utility, the experiment sessions shall be organized into a folder structure.

If we recall Figure 3.8, we have only covered the `Measurement` and `Entry` parts. There is still missing a way to represent sensor outputs. One option is to add them to the

`MeasurementObject`, or to create a separate container for them. We shall account for both cases, letting the user specify the data in both the object and in separate data variables.

■ Sensor outputs

A physical setup, as the one we have in Section 3.1.1 can have multiple sensors. Each sensor can have several output types, each with different representations, such as a list or binary data in the form of a file.

An unambiguous representation thus requires a sensor output instance to contain the sensor where it comes from, the output quantities and the data corresponding to each quantity.

- Sensor output
 - Sensor
 - Quantity 1: Values 1
 - Quantity 2: Values 2
 - ...

Logically, a particular setup is then a collection of sensors—setup elements.

■ Grasp; Grasp proposal

Next, we shall, for the time being, assume that a grasp was made by a gripper that has a set geometry and a single degree of freedom for its moving components. A grasp is then at the very minimum represented by a position, orientation and a “grasped” boolean attribute.

A grasp is connected to a measurement, however a measurement does not generally need a grasp to be successful: for instance when estimating an object’s material properties through sound, a gripper can be used to merely tap an object to get a sound recording from it. Yet, it still makes a difference where the object was tapped, since the sound can be different depending on the place on the object. For example a water bottle has a hard cap and a softer, hollow body.

If we want to add more complicated grippers, such as those with more than one degree of freedom, we have to take into account any number of possible configurations. Furthermore, it is unknown at that point what other properties a **grasp** contains.

A common, almost universal, property of a grasp is also the speed at which the gripper closes. It shall also be added. The closing speed is assumed to be constant. Another property of a grasp that is important for grasping is the maximum force used during the grasp. It can also be viewed as the force necessary to perform the grasp.

If providing the force in Newtons is not possible, there will be added an accompanying `units` field to specify the units that the grasp effort will be represented in. When left

empty, it shall be assumed that the units are proprietary to the sensor. For example, a user enters the number 0.5 for the gripper Robotiq 2F85 from the setup in Section 3.1.1. The units are not clear, but in the context of the 2F85 gripper it is clear that it means 50% of the maximum because the interval of values that can be sent into it is $[0, 1]$.

Hence, the following structure of a grasp:

- Grasp
 - Rotation [xyz]
 - Translation [xyz]
 - Grasped [yes/no]
 - Closing speed
 - Maximum effort
 - Units

■ Object pose

Related to successful grasping is the ability of a robotic arm to get into the necessary position to grasp an object. An algorithm may output a grasp with relation to an object, but in case it does not take into account the reachability of a pose in real setup. To address this, we shall also save the object's pose with relation to the robot manipulator base. This gives us information about the feasibility of a grasp in a real situation.

- Object pose
 - Rotation [xyz]
 - Translation [xyz]

■ Object instance

Let us now take a closer look at the measurement structure as depicted in Section 3.2.1. There, we have already established that some measurements are provided by humans, namely the `dataset`, `dataset id`, `maker` and `common name` and that they are members of an object instance.

The object instance is intended to act as a proxy for a real object to which then belong the measurements. In case one-time fields of unspecified format need to be added to the object instance, an additional JSON field named `other` will be present as well.

- Object Instance
 - Dataset
 - Identifier in dataset

- Maker
- Common name
- Other—JSON
- File

■ User

We want to keep track of who uploaded which measurement, so a user is going to be associated with the `object` instance that they measured, the `measurement` of that instance and the property estimation, called `entry`. The relational graph depicting the links between these elements is found in Figure 4.2.

■ Experiment structure

By default it shall be assumed that a session consists of one `experiment` and multiple measurements, each with its own property estimation. Meanwhile sensor outputs shall be captured in variables visible in scope to the `Butler` class after the function that generates the data variables finishes. The variables will have to be convertible to the JSON format in Python. A function to help with the conversion of common data types will be provided. For now the experiment structure is provided in Figure 3.10.

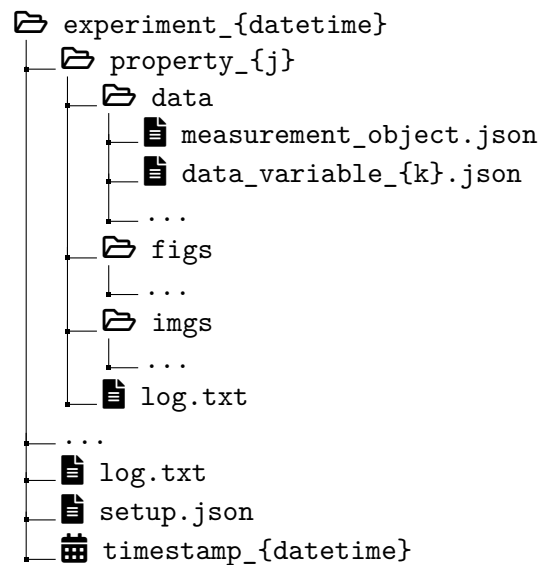


Figure 3.10: The directory structure as generated by Butler.

Let us take the top-level element first

- `experiment_{datetime}`: This is the encompassing folder containing one experiment, the `datetime` variable field is the timestamp in the form of `YYYY_MM_DD_HH_mm_ss`.

In it, there is an arbitrary number of properties and some files describing the experiment instance.

- `property_name_j`: This is the directory where the measurement for one property is located. j is indexed from 0.
- `log.txt`: This is a top level log file which will contain all print outputs of the function that has been marked to be logged.
- `setup.json`: This should contain a dictionary mapping the type of the setup element to its name. For example,

```
{"gripper": "robotiq_2f85", "arm": "kinova_gen3", ...}
```

The location of the `setup.json` is required in `Butler`'s arguments when initializing it.

- `timestamp`: This is a timestamp in the same format as the experiment directory name—YYYY_MM_DD_HH_mm_ss.

Inside the property there are three directories and another log file.

- `log.txt`: This one contains only the printed lines that contain selected **keywords**. The reason for this split is that these prints, containing for example the sequence [STIFFNESS], can be related to the measurement being performed enabling prints irrelevant to the measurement to be ignored.
- `data`: This should contain measurement data and sensor outputs.
- `figs`: This should contain generated figures.
- `imgs`: This should contain any images that might have been taking during the measurement.

Finally, we have the data directory.

- `measurement.json`: This is the JSON file containing the property estimation and can also contain some sensor outputs.
- `data_variable_{k}.json`: Each sensor output may have one JSON file dedicated to it. For example: `{"robotiq_2f85": "position": [...], "current": [...], "time": [...]}` .

This data is now in JSON format, thus the remaining step is putting it all together into a request to the server.

■ Uploading

After data is gathered from the local setup, it is then necessary to upload it to the server, see the second role of `Butler` in Figure 3.9. For that a pipeline is going to be used to convert the experiment folders to a more presentable format.

This process again is split up into two main parts:

- Processing of the experiment directory into one coherent JSON file
- Uploading, which itself is split into three parts:
 - File extraction from the paths provided in the formatted JSON file
 - Final JSON formatting—removing all references of absolute file paths
 - Request creation—specifying the request’s JSON data and files

■ 3.4.2 Butler implementation

In Python, the easiest way to mark a function to be logged is in the form of a decorator. Therefore, the Butler function decorator was created. The source code is accessible at <https://github.com/Hartvi/butler>. A decorator is generally a function wrapper that can be added above a function in the source code like so:

```
def decorator(f):
    def wrapper(*args, **kwargs):
        print("Running a decorated function")
        return f(*args, **kwargs)
    return wrapper

@decorator
def func(arg1, arg2):
    ...
```

The decorator then takes the decorated function as an argument and returns a function that takes the same arguments as the original function. This way, the functionality of any function can be extended using a decorator. One can prepend or append custom functionality and read the inputs or outputs of the function. However, the function itself remains a black box.

Combining the built up structures from the previous sections, the inputs to Butler are going to be the following:

```
class Butler(keywords=(),
             keep_keywords=True,
             setup_file="setup.json",
             read_return=True,
             session_parent_dir=config.experiment_directory,
             output_variable_name="",
             data_variables=(),
             img_files=(),
             fig_files=(),
             data_files=(),
             ignore_colours=True,
             create_new_exp_on_run=False)
```

Note, this structure arose while taking into account the requirements of the action selection Bayesian inference framework which is being developed in parallel to the continuation of [17].

Let us now examine the arguments one by one:

- **keywords**: List of keywords whose lines will be extracted when printed.
- **keep_keywords**: Whether to also save the keywords with the rest of the printed line.
- **setup_file**: Path to the JSON containing the setup mappings. E.g. `{"gripper": "robotiq_2f85", ...}`
- **read_return**: Whether to take the return value (or first element in the returned tuple) as the measurement output.
- **session_parent_dir**: Directory where to save the experiments; the default is specified in the config file.
- **output_variable_name**: The string name of the variable that contains the data that is otherwise by default assumed to be returned by the decorated function. Has to be visible in the scope where the decorated function is called. E.g. `self.data_var` or `just_data_var`.
- **data_variables**: The sensor output variables. Format: `{"source_sensor_name": {"quantity (e.g. position)": [list, of, values], ...}, ...}`
- **img_files**: A list of file paths that will be copied to `experiment_i/property_j/imgs` every time the function is run.
- **fig_files**: A list of file paths that will be copied to `experiment_i/property_j/figs` every time the function is run.
- **data_files**: A list of file paths that will be copied to `experiment_i/property_j/data` every time the function is run.
- **ignore_colours**: Whether to omit writing special colour characters in the log files.
- **create_new_exp_on_run**: Whether to create a new `experiment_i` folder on every run of the function.

A need that arose after creating the initial draft of **Butler** as seen above was the option to add temporary data/files that exist only for one run of the decorated function. This is also partly due to the fact that the function is a black box and it is uncertain whether its outputs have a constant structure.

To that end, functions modifying **Butler**'s behaviour that can be called from inside the decorated function and that have a temporary effect for only one run of the decorated

function are necessary. These need to be able to be called without affecting its execution while at the same time providing additional context for that run.

The first function is called `add_object_context`. If the user is measuring an object whose identity they themselves know and they want to add it as extra information to the measurement, then they call this function, specifying the `name`, `maker`, `dataset` and the `dataset_id`—the ID of the object in the dataset. These fields are those described in Section 3.2.1.

```
def add_object_context(context, override_recommendation=False)
```

The arguments have the following meanings.

- `context`: This is the object context entered by the user. The format is the following: `{"maker": "ikea", "common_name": "wineglass", "dataset": "ycb"}`. If `"dataset_id"` is present, `"dataset"` must also be present.
- `override_recommendation` Whether to remove the constraint that the context keys have to be one of `["maker", "common_name", "dataset", "dataset_id"]`.

The second function serves as a temporary alternative to the arguments `img_files`, `fig_files`, `data_files`. Whereas the arguments set the file paths for every run of the function, calling `add_tmp_files` inside the decorated function enables the target file paths to be different every run.

```
def add_tmp_files(file_paths, tmp_file_folder, target_names=None)
```

The arguments have the following meanings:

- `file_paths`: A list of file paths that is to be copied to the generated experiment folder.
- `tmp_file_folder`: This is one of the three names: `"data"`, `"figs"`, `"imgs"`, designating which of these three folders the source paths `file_paths` are to be copied into.
- `target_names`: This may be filled with a list with the same length as the `file_paths` argument. Each source file path's file name is then copied into the new name in the `tmp_file_folder` directory.

The third auxiliary function that is used for recording experiments is `add_measurement_img`. The reason for adding extra images in a measurement is for other users to be able to visually confirm what object was measured or how the measurement was performed depending on what the uploader recorded.

```
def add_measurement_img(img_path)
```

The only argument for this function is the image path.

- `img_path`: The path to the photo of the object being measured. It will be saved as `img.{suffix}` in the `data` folder.

Chapter 4

Experiments and Results

This chapter consists of the logging the measurements of a subset of objects from the YCB dataset and objects chosen for their visual similarity to those YCB objects. Subsequently these measurements and property estimations gained from the measured data will be uploaded to the database.

The process on the side of the user is as follows:

1. Data gathering
2. Formatting
3. Uploading

The server does the following tasks:

1. Data validation
2. Saving to database



Figure 4.1: Selected YCB objects and objects visually similar to them.

The server and data gathering scripts will be tuned according to the needs of the real setup. The measurements will initially be performed by manually setting the pose of the

object and of the gripper to reliably obtain grasps and measurements. Afterwards to gather more grasp proposals, the GPD grasping net [25] shall be employed to gather more diverse grasps. These two above methods will ensure that the objects' properties are going to get measured as well as grasps being generated. The set of objects to be measured is depicted in Figure 4.1.

Since this work is being created in tandem with the exploratory action selection framework [17], the objects being measured have been chosen for their ambiguous properties. For example, the two sponges and 9-hole peg box in Figure 4.1 have been chosen for their similar appearances yet different properties. The wooden box for example is much heavier and much stiffer than the two sponges. The two sponges between each other have different stiffnesses despite being almost identical in appearance.

4.0.1 Database structure

Given the knowledge built up in Section 3.2 the database has been designed in Django with its structure depicted in Figure 4.2. Some additional fields were added in the attempt to wholly cover hypothetical scenarios not possible at the lab, such as `Other` fields in `Object Instance`, `Grasp` for files or JSON data.

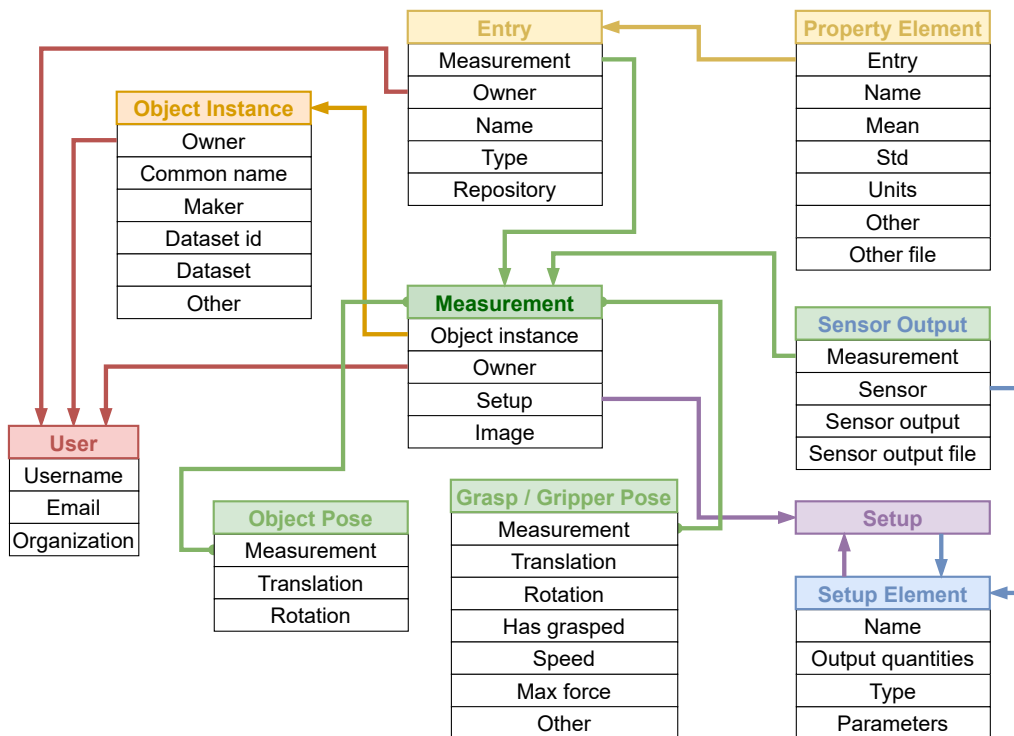


Figure 4.2: Implementation of the database's structure.

4.0.2 Measurement

Measuring was performed on the physical setup from Figure 3.1. For the measurement `Butler` from Section 3.4.2 was used.

■ Compatibility issues

Some compatibility issues arose when trying to insert `Butler` into the main script. The biggest one of them was discrepancies between Python2.7 and Python3 versions. To solve this, the decorator was rewritten in neutral Python compatible with both versions of Python.

■ Recording experiments

To record an experiment on the setup at the lab the function `exploratoryAction` was decorated like so:

```
@Butler(keywords=['[BUTLER-TEST]', '[INFO]',
                 '[INFER-INFO]', '[INFER]',
                 '[ACSEL]', '[ACSEL-INFO]'],
        setup_file="/home/robot3/vision_ws/src/.../butler/setup.json",
        data_variables=('self.data_variables'))
def exploratoryAction(self, planned_action, mod_specs,
                    translation_pos, mes_rot, iteration, ID):
```

What this does is save all prints coming from this function that contain one or more of the keywords, it copies the `setup.json` into the experiment folder, saves the `self.butler_values` variable into a file called `data_variables.json` and saves the first return value of the decorated function into `measurement.json`.

The format of the return value has been hitherto uncertain but based on the database structure it is expected to be roughly in the format as shown below. Later on in the formatting stage the lexicological roots of the names of the members as enumerated below are going to be recursively searched for and matched with regular expressions.

```
class PropertyMeasurement:
    # e.g. mass, elasticity, vision, sound
    property_name
    measurement_type # continuous, discrete
    parameters/prediction/output
    units
    grasp/gripper_pose
    object_pose
    values
    repository
    other
    other_file
```

The `data_variables` variable contains the dictionary:

```
{"camera": {"image": "img_cam1.png"}}
```

At the lab specific objects were measured one by one, so in this case the function `add_object_context` can be used to add extra information. For example, when we are measuring the plastic banana from the YCB dataset we call the following *inside* the decorated `exploratoryAction` function:

```
def exploratoryAction(...):
    ...
    context_dict = {"dataset_id": "011_banana",
                   "dataset": "ycb",
                   "common_name": "banana_ycb"}
    Butler.add_object_context(context_dict)
    ...
```

This results in an `object_context.json` containing:

```
{"common_name": "banana_ycb", "dataset_id": "011_banana", "dataset": "ycb"}
```

The resulting directory structure after running a measurement with the above modifications is depicted in Figure 4.3 for a object category vision measurement.

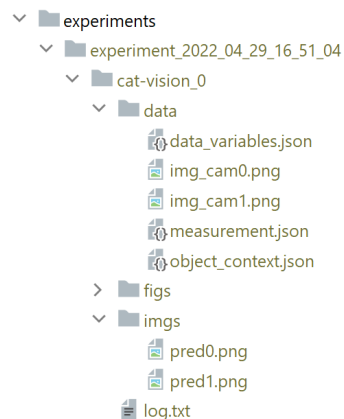


Figure 4.3: An example directory structure of a vision category measurement.

4.0.3 Physical measurement

To obtain grasps it is necessary to have some frame of reference of the objects and to know the gripper pose. This was achieved with CosyPose [26] trained and setup at the lab by Jan Behrens from the Czech Institute of Information Robotics and Cybernetics (CIIRC).

CosyPose estimates the 6D pose of pretrained objects from an input RGB image. In our case, CosyPose was trained only on a subset of 21 YCB objects called YCBV [27]. One such object is the plastic cup with the labels `065-f_cups`, whose measurement can be seen in Figure 4.4.

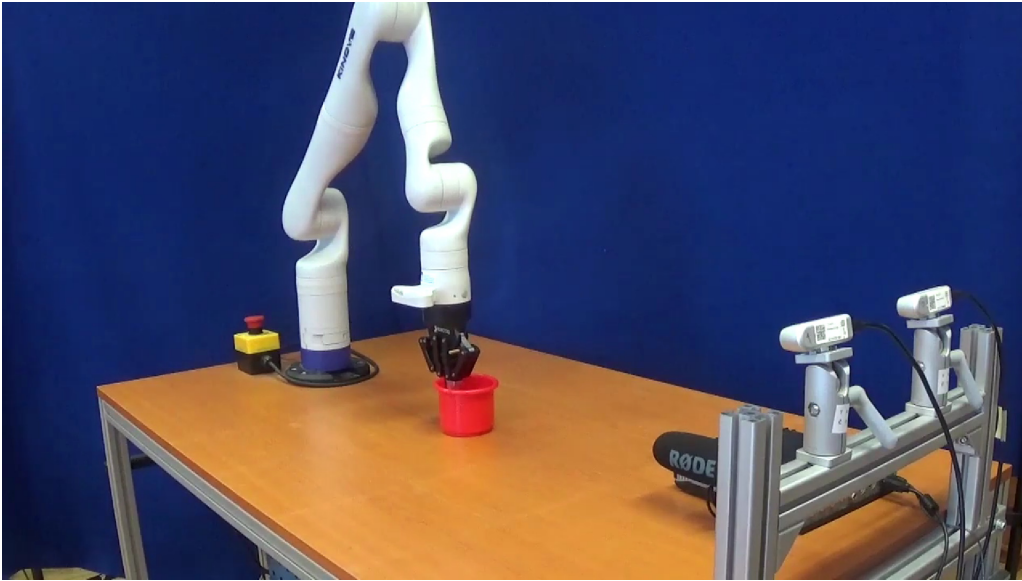


Figure 4.4: Stiffness measurement of the plastic cup from the YCB dataset.

■ Grasp generation

One of the goals of this work is to generate grasp proposals. To that end, Lukáš Rustler helped with getting PointNetGPD [25] up and running on the local setup. Due to random noise issues in the point cloud generated by the Intel D435 depth cameras, the grasp generation was only able to be used on soft objects.

One caveat of this configuration is that, for example, for the sponges from Figure 4.1 there is no general consensus on their coordinate systems. That is why there were arbitrarily—not randomly, which means the edges of the objects line up with the axes of their coordinate systems—coordinate systems assigned to the objects. Just like the `add_measurement_img` functionality from Section 3.4.2, images of how the objects were positioned relative to the robot base were added in post-processing, see Section 4.0.4.

■ 4.0.4 Formatting

After the experiments have been saved to disk, it is time to compile the experiments into a single uploadable JSON dictionary. For that a module was created to search the experiment and property folders for JSON files and files on disk that are pointed to in the value fields in `measurement.json` and data variable JSONs.

For each property directory—in Figure 4.3 it is `cat-vision_0`—there is created one *almost* uploadable JSON file that has now standardized keys that the uploader module can then use to access the fields.

The formatting module checks the property directory for permissible variations of properties that are necessary to make the property entry uploadable.

■ Post-processing

Sometimes—due to human error or otherwise—some mistakes are introduced or fields left out from the measurement. If these mistakes are in the form of, for example missing `dataset` in the `object_context.json`, then there is a function for that as well.

```
def change_experiment_jsons(update_dict,
                            experiment_directory,
                            json_file_name,
                            rule,
                            replace=False):
```

What this does is it updates any JSON file named `json_file_name` located in the `experiment_directory` in place with the `update_dict` argument if the path to the file `json_file_name` fulfills a rule set by `rule`.

```
change_experiment_jsons({"repository": "https://github.com/user/repo"},
                        "path/to/experiment_2022_04_29_16_51_03",
                        "measurement.json",
                        lambda x: "at-vision" in x and "data" in x)
```

The above call of the function is intended to update all paths that look like this: `path/to/experiment_2022_04_29_16_51_03/*at-vision*data*/measurement.json` with the mapping `{"repository": "https://github.com/user/repo"}`. This is equivalent to updating the JSONs like python dictionaries:

```
measurement["repository"] = "https://github.com/user/repo"
```

■ 4.0.5 Uploading

The formatted dictionaries are then loaded and processed right before uploading, removing all absolute path references whilst leaving only the base names of the files. The files pointed to in the formatted dictionaries are opened and loaded into memory. Once these two steps are done, the request can be made.

■ Server interface

Interfacing with the server is done through the Python `requests` library. An example of this is as follows:

```
req = requests.request(method="POST",
                       url=url, auth=auth_tuple,
                       data=data_dict, files=file_bytes)
```

Downloading is simply done by reading the website text and loading it as a JSON file.

```
r = requests.request("GET", "http://www.cvut.cz/")
data_dict = json.loads(r.text)
```

Downloading will be left for the user to handle.

■ Implementation

The final function that we use to upload data is `lazy_post_measurements`.

```
def lazy_post_measurements(auth_tuple, url)
```

- `auth_tuple`: This is a tuple in the form ("username", "password").
- `url`: This is the URL to post to. For the local client it is `127.0.0.1/rest/[measurements, entries]`

This function expects the previous steps from the processing pipeline to have already been taken. The lack of other arguments is due to the fact that the experiments' and formatted dictionaries' locations are expected to be defined in a `config.py` file which is located in the same directory as `butler.py`.

When called, it uploads whatever JSON file has not yet been uploaded and returns a dictionary showing which files have been uploaded successfully.

```
{"path/to/formatted_experiment.json": true/false, ...}
```

■ 4.0.6 Database

Let us now examine what the database and its website looks like. The back-end functionality is already accessible with scripts but we also want a user-friendly front-end with links to all of the components of the database and documentation on how to use it. The website is going to be deployed on <https://ptak.felk.cvut.cz/ipalm/> and may be visible on <http://bayes.felk.cvut.cz> as well. The database's source code is in the repository https://github.com/Hartvi/object_database.

■ Homepage

The homepage for the database is a crossroad to different sub-applications, see Figure 4.5. The three semantic sections are the Django database, the Django Rest Framework (DRF) and the offline experiment logger and uploader.

The DRF homepage is in Figure 4.6. There we can see the publicly visible models and the links to the lists of

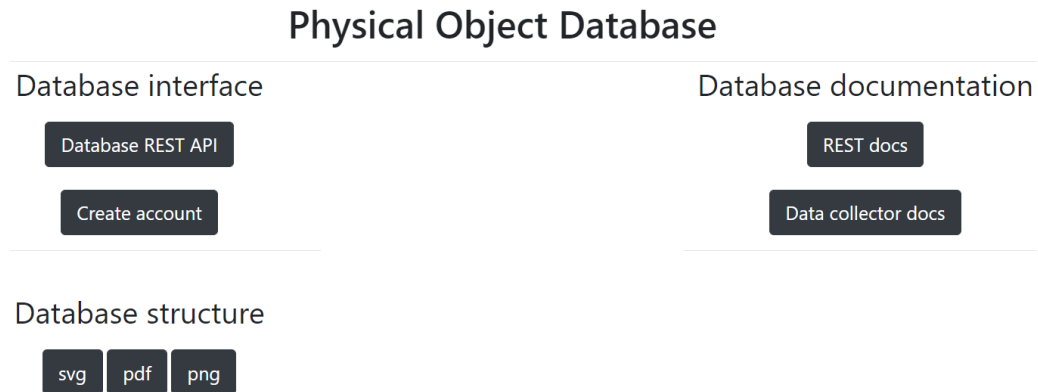


Figure 4.5: The website homepage. The top left is the Django Rest Framework (DRF) home and user register form, below is the internal database structure. On the right from the top there is the automatically generated interfacing documentation from the DRF and below there is the documentation for the experiment logging and uploading module.

■ Database REST API (Django Rest Framework)

The Django Rest Framework conveniently provides a user interface. The appearance of a measurement entry can be seen in Figure 4.7.

■ Create account

This redirects the user to a create user form. One can only post to the server with an account.

■ Database structure

The database structure has already been shown in Figure 4.2 The database, as displayed when the link is clicked, otherwise has additional internal Django-specific models and fields.

■ REST docs

These docs are automatically generated by DRF and only show how to interact with individual components of the database.

■ Data collector docs

The documentation for the offline module `butler` was generated from `docstrings` using the Sphinx automatic documentation tool [28]. The top of the documentation page can be seen in Figure 4.10.

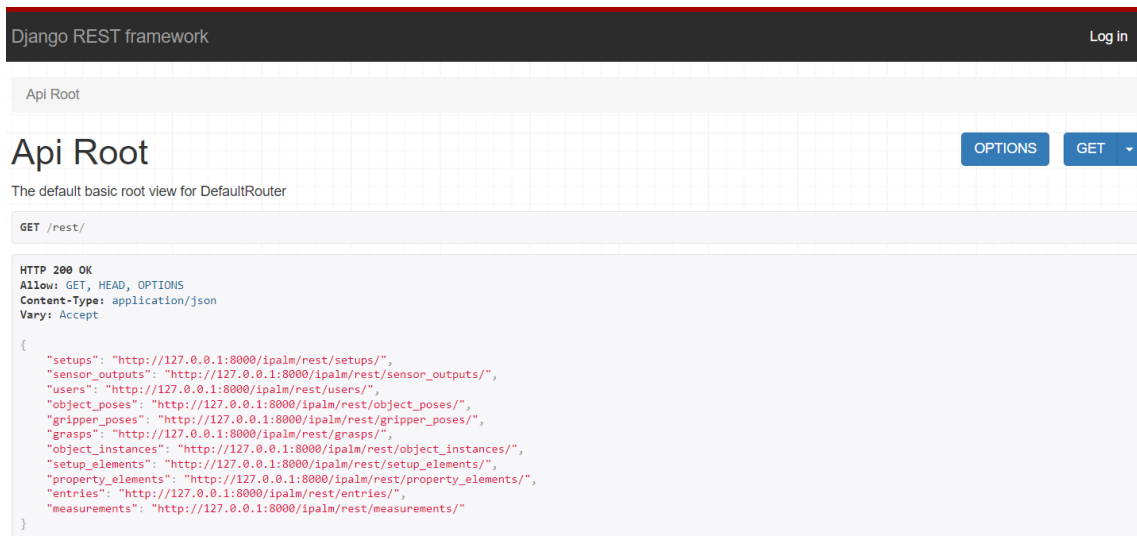


Figure 4.6: The DRF homepage with links to lists of the main data models from the database.

Benchmarking

How do we compare the methods of obtaining object properties? In many cases, there is no ground truth measurement of a property. Even if there were a ground truth measurement such as a professional stiffness measurement of, say, the YCB mustard bottle, it may not be comprehensive enough to describe the stiffness of every part of the bottle; its stiffness would also depend on its contents, whether it is open, etc. A professional setup may measure stiffness with flat plates pushing on the mustard bottle, yet a gripper may grasp it in a way impossible to do with conventional professional setups.

For that reason a complete description of the measurement is necessary. Grippers with the same shape may even produce different output quantities: one of them might output the gripper's motor current while the other one has a flat pressure sensor mounted on its pads.

One option for comparing the quality of property estimates is comparing the algorithms based on the source data type used for the estimations. For example, in the BOP challenge [29] contestants are divided into groups based on the type of input to their object pose estimation algorithms. There are separate leaderboards for RGB, RGBD and depth-only algorithms. Another metric is the speed at which the algorithms produced their results.

Django REST framework Log in

Api Root / Measurement List / Measurement Instance

Measurement Instance OPTIONS GET

GET /rest/measurements/45/

```

HTTP 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
  "url": "http://127.0.0.1:8000/rest/measurements/45/",
  "owner": "http://127.0.0.1:8000/rest/users/1/",
  "setup": "http://127.0.0.1:8000/rest/setup/7/",
  "object_instance": "http://127.0.0.1:8000/rest/object_instances/33/",
  "png": null,
  "entries": [
    {
      "url": "http://127.0.0.1:8000/rest/entries/77/",
      "property_element": [
        {
          "url": "http://127.0.0.1:8000/rest/property_elements/742/",
          "entry": "http://127.0.0.1:8000/rest/entries/77/",
          "name": "clamp",
          "value": 0.0,
          "std": null,
          "units": "clamp",
          "other": "[]",
          "other_file": null
        }, ... shortened for clarity
      ]
    },
    {
      "measurement": "http://127.0.0.1:8000/rest/measurements/45/",
      "owner": "http://127.0.0.1:8000/rest/users/1/",
      "created": "2022-05-02T08:37:27.860093Z",
      "repository": "https://github.com/hartvjr/detector2",
      "type": "categorical",
      "name": "cat-vision"
    }
  ],
  "grasp": null,
  "object_pose": {
    "url": "http://127.0.0.1:8000/rest/object_poses/7/",
    "measurement": "http://127.0.0.1:8000/rest/measurements/45/",
    "rx": 173.309,
    "ry": 15.525,
    "rz": -21.009,
    "tx": 0.465,
    "ty": -0.013,
    "tz": 0.031
  },
  "gripper_pose": {
    "url": "http://127.0.0.1:8000/rest/gripper_poses/6/",
    "measurement": "http://127.0.0.1:8000/rest/measurements/45/",
    "rx": -180.0,
    "ry": 0.0,
    "rz": 0.0,
    "tx": 0.402437210083,
    "ty": -0.01078241612488151,
    "tz": 0.160072049689,
    "grasped": true
  },
  "sensor_outputs": [
    {
      "url": "http://127.0.0.1:8000/rest/sensor_outputs/84/",
      "sensor": "http://127.0.0.1:8000/rest/setup_elements/11/",
      "sensor_output_file": "http://127.0.0.1:8000/media/img_cam1_MNwTLn.png",
      "sensor_output": "{\"image\": \"img_cam1.png\"}",
      "measurements": "http://127.0.0.1:8000/rest/measurements/45/"
    }
  ],
  "created": "2022-05-02T08:37:27.926852Z"
}

```

Figure 4.7: An uploaded categorical measurement as it appears rendered by the Django Rest Framework. Only one category has been left in for brevity.

Registration

Username*

Email*

Organization*

Password*

Confirm password*

[Register](#)

Figure 4.8: The register form to create a user.

Online physical properties REST API

- [ipalm](#)
- [rest](#)

Online physical properties REST API

A Web API for realtime uploading/downloading data for physical properties.

ipalm

rest > entries > list

[GET](#) /ipalm/rest/entries/ Interact

Query Parameters

The following parameters should be included as part of a URL query string.

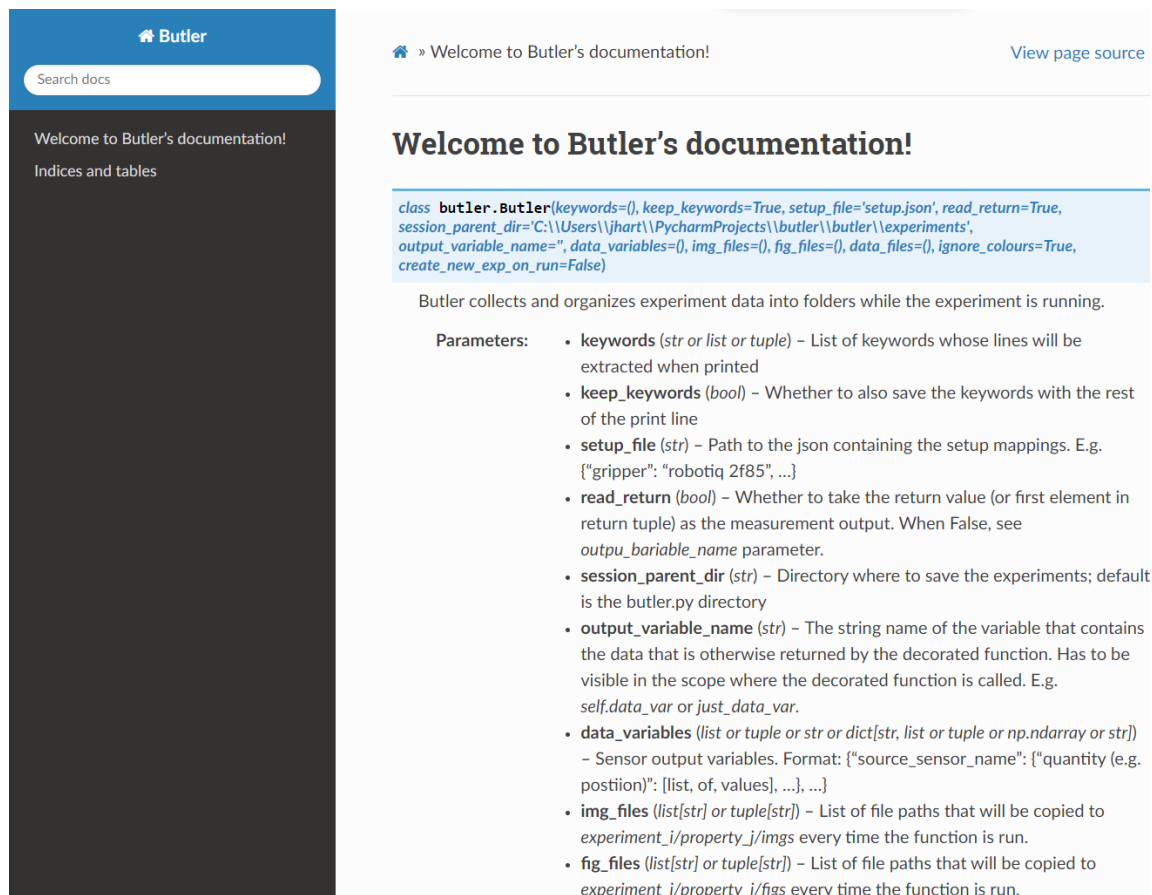
Parameter	Description
page	A page number within the paginated result set.

```
# Install the command line client
$ coreapi get http://127.0.0.1:8000/ipalm/docs/

# Load the schema document
$ coreapi get http://127.0.0.1:8000/ipalm/docs/

# Interact with the API endpoint
$ coreapi action ipalm rest entries list -p page=...
```

Figure 4.9: Automatically generate DRF documentation.



Butler

Search docs

Welcome to Butler's documentation!
Indices and tables

» Welcome to Butler's documentation! [View page source](#)

Welcome to Butler's documentation!

```
class butler.Butler(keywords=(), keep_keywords=True, setup_file='setup.json', read_return=True,
session_parent_dir='C:\\Users\\jhart\\PycharmProjects\\butler\\butler\\experiments',
output_variable_name="", data_variables=(), img_files=(), fig_files=(), data_files=(), ignore_colours=True,
create_new_exp_on_run=False)
```

Butler collects and organizes experiment data into folders while the experiment is running.

Parameters:

- **keywords** (*str* or *list* or *tuple*) – List of keywords whose lines will be extracted when printed
- **keep_keywords** (*bool*) – Whether to also save the keywords with the rest of the print line
- **setup_file** (*str*) – Path to the json containing the setup mappings. E.g. {"gripper": "robotiq 2f85", ...}
- **read_return** (*bool*) – Whether to take the return value (or first element in return tuple) as the measurement output. When False, see *output_variable_name* parameter.
- **session_parent_dir** (*str*) – Directory where to save the experiments; default is the butler.py directory
- **output_variable_name** (*str*) – The string name of the variable that contains the data that is otherwise returned by the decorated function. Has to be visible in the scope where the decorated function is called. E.g. *self.data_var* or *just_data_var*.
- **data_variables** (*list* or *tuple* or *str* or *dict*[*str*, *list* or *tuple* or *np.ndarray* or *str*]) – Sensor output variables. Format: {"source_sensor_name": {"quantity (e.g. position)": [list, of, values], ...}, ...}
- **img_files** (*list*[*str*] or *tuple*[*str*]) – List of file paths that will be copied to *experiment_i/property_j/imgs* every time the function is run.
- **fig_files** (*list*[*str*] or *tuple*[*str*]) – List of file paths that will be copied to *experiment_i/property_j/figs* every time the function is run.

Figure 4.10: The documentation page generate using Sphinx.

Chapter 5

Conclusion, Discussion, and Future Work

In this work we have discussed how previous works relate to our goal and what functionalities they bring to the table. It was established that there is space for a database that would enable different datasets to live in the same data structures. To that end, an interactive database was created in Django and Django Rest Framework to enable users to freely upload and download measurement and property estimation data. To relieve the user of the burden of formatting the data specifically for the server, a logging module called `Butler` was created. `Butler` is a function decorator which means that in the user's code it can be added above the function that outputs the measurement and property estimation data.

The `Butler` module was tested on the local setup and a subset of the YCB dataset and some other common household objects were measured at the lab. The objects frames of reference were generated using `CosyPose` and using the knowledge of the pose of the gripper and of the objects, grasps were generated and added to the measurements. This way, grasps, in addition to other measurements, were added to the database's repertoire.

There was also discussed a methodology of comparing the performances of individual setups and algorithms. One option is comparing the source data type. For example in the BOP challenge [29] the rankings are split by the type of input the algorithms used: RGB, RGBD or depth-only. Another metric is the speed at which the algorithms produced their results.

The database as created with Django is able to be automatically migrated, which means that it has an updatable structure. In theory the database could then encompass all manners of robotic experiments, provided there is enough disk space and an interface to upload them.

The limitation at this point is the lack of features. As of now the database is just raw structured data storage. Another issue might be ease of use, as the `Butler` module and uploading was only tested on the local setup, albeit in Python versions 2 and 3.

In the future testing the `Butler` module on other setups would provide valuable feedback on how to make it easier to use, improve its functionality and clarify the documentation. Furthermore, it would help to extend the database to be able to represent different gripper configurations and perhaps even derive a general representation of deformable objects, such as cloths.

If there are at some point in the future hundreds or thousands of measurements saved in



Bibliography

- [1] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “Imagenet: A large-scale hierarchical image database,” in *2009 IEEE conference on computer vision and pattern recognition*. IEEE, 2009, pp. 248–255.
- [2] A. Krizhevsky, V. Nair, and G. Hinton, “Cifar-100 (canadian institute for advanced research).” [Online]. Available: <http://www.cs.toronto.edu/~kriz/cifar.html>
- [3] P. Stoudek and M. Mareš, 2020. [Online]. Available: <https://gitlab.fel.cvut.cz/body-schema/ipalm/ipalm-grasping>
- [4] R. Dahl, *Charlie and the Chocolate Factory*. Puffin Books, 1964.
- [5] B. Calli, A. Singh, A. Walsman, S. Srinivasa, P. Abbeel, and A. M. Dollar, “The ycb object and model set: Towards common benchmarks for manipulation research,” in *2015 international conference on advanced robotics (ICAR)*. IEEE, 2015, pp. 510–517.
- [6] F. Bottarel, G. Vezzani, U. Pattacini, and L. Natale, “Graspa 1.0: Graspa is a robot arm grasping performance benchmark,” *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 836–843, 2020.
- [7] D. Morrison, P. Corke, and J. Leitner, “Egad! an evolved grasping analysis dataset for diversity and reproducibility in robotic manipulation,” *IEEE Robotics and Automation Letters*, vol. 5, no. 3, pp. 4368–4375, 2020.
- [8] J. Mahler, J. Liang, S. Niyaz, M. Laskey, R. Doan, X. Liu, J. A. Ojea, and K. Goldberg, “Dex-net 2.0: Deep learning to plan robust grasps with synthetic point clouds and analytic grasp metrics,” 2017.
- [9] M. Thosar, C. A. Mueller, G. Jäger, J. Schleiss, N. Pulugu, R. Mallikarjun Chennaboina, S. V. Rao Jeevangekar, A. Birk, M. Pfingsthorn, and S. Zug, “From multi-modal property dataset to robot-centric conceptual knowledge about household objects,” *Frontiers in Robotics and AI*, vol. 8, p. 87, 2021.
- [10] M. Ciocarlie, “Ros household objects,” 2014. [Online]. Available: <http://wiki.ros.org/household%20objects>
- [11] M. Vincze, M. Rudorfer, M. Suchi, I. G. Camacho, J. Borras, G. Alenya, A. Leonardis, M. Srihadran, A. Alliegro, and T. Tommas, “Benchmarks for understanding robotic grasping,” 2022. [Online]. Available: <https://www.acin.tuwien.ac.at/project/burg/>

- [26] Y. Labbé, J. Carpentier, M. Aubry, and J. Sivic, “CosyPose: Consistent multi-view multi-object 6d pose estimation,” in *European Conference on Computer Vision*. Springer, 2020, pp. 574–591.
- [27] Y. Xiang, T. Schmidt, V. Narayanan, and D. Fox, “Posecnn: A convolutional neural network for 6d object pose estimation in cluttered scenes,” *arXiv preprint arXiv:1711.00199*, 2017.
- [28] G. Brandl, “Sphinx documentation,” 2022. [Online]. Available: <https://www.sphinx-doc.org/en/master/>
- [29] T. Hodaň, M. Sundermeyer, B. Drost, Y. Labbé, E. Brachmann, F. Michel, C. Rother, and J. Matas, “BOP challenge 2020 on 6d object localization,” in *European Conference on Computer Vision*. Springer, 2020, pp. 577–594.
- [30] T. Salman, R. Jain, and L. Gupta, “Probabilistic blockchains: A blockchain paradigm for collaborative decision-making,” in *2018 9th IEEE Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*. IEEE, 2018, pp. 457–465.

I. Personal and study details

Student's name: **Hartvich Jiří** Personal ID number: **483636**
Faculty / Institute: **Faculty of Electrical Engineering**
Department / Institute: **Department of Cybernetics**
Study program: **Cybernetics and Robotics**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Estimating Object Properties Through Robot Manipulation - Dataset and Benchmark

Bachelor's thesis title in Czech:

Odhadování fyzikálních vlastností objektů pomocí robotické manipulace – dataset a srovnávací metriky

Guidelines:

While progress in computer vision has been largely fueled by shared datasets and benchmarks, advances in robotics in general, and in robot manipulation in particular are slower, in part due to the unavailability of shared data and protocols. The project IPALM (Interactive Perception-Action-Learning for Modelling Objects, <https://sites.google.com/view/ipalm>) aims to fill this gap by providing methods, datasets and benchmarks focused on automatically extracting physical object properties like stiffness, mass, or surface roughness, which can be subsequently used for manipulation.

1. Survey existing datasets and benchmarks related to robot manipulation (e.g., [1][2][4]).
2. Design software that meets the following criteria:
 - a. Database of categories of every-day objects and prior probability distributions of their properties (e.g. a mug is typically from ceramic, has a mean mass of 100 g, etc.). The database can be seeded from [2].
 - b. Interface the database with software that explores a particular object instance in front of a robot camera and manipulator - following up on [3]. Exploratory actions will be selected and new measurements will be added to the database with appropriate tags (e.g., which gripper and which parameters were used).
 - c. Grasp proposals or means of obtaining them will be part of the database.
3. Create a pilot dataset using the objects available at the lab (e.g., YCB dataset [2]).
4. Prepare a protocol for benchmarking - comparing the performance of different robot setups and algorithms in finding object properties. Evaluate the framework of [3] and create a first entry in a leader board.
5. Prepare and interface for expansion of the database, with the possibility of active participation of contributors from the outside.

Bibliography / sources:

- [1] Bottarel, F., Vezzani, G., Pattacini, U., & Natale, L. (2020). GRASPA 1.0: GRASPA is a robot arm grasping performance benchmark. *IEEE Robotics and Automation Letters*, 5(2), 836-843. <https://github.com/robotology/GRASPA-benchmark>
- [2] Calli, B., Singh, A., Walsman, A., Srinivasa, S., Abbeel, P., & Dollar, A. M. (2015). The YCB object and model set: Towards common benchmarks for manipulation research. In 2015 International Conference on Advanced Robotics (ICAR) (pp. 510-517). IEEE. <https://www.ycbbenchmarks.com/>
- [3] Kružíliak, A. (2021), 'Exploratory action selection to learn object properties through robot manipulation', Bachelor thesis, Faculty of Electrical Engineering, Czech Technical University in Prague.
- [4] Morrison, D., Corke, P., & Leitner, J. (2020). EGAD! an Evolved Grasping Analysis Dataset for diversity and reproducibility in robotic manipulation. *IEEE Robotics and Automation Letters*, 5(3), 4368-4375. <https://dougsm.github.io/egad/>

Name and workplace of bachelor's thesis supervisor:

Mgr. Mat j Hoffmann, Ph.D. Vision for Robotics and Autonomous Systems FEE

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **17.01.2022** Deadline for bachelor thesis submission: **20.05.2022**

Assignment valid until: **30.09.2023**

Mgr. Mat j Hoffmann, Ph.D.
Supervisor's signature

prof. Ing. Tomáš Svoboda, Ph.D.
Head of department's signature

prof. Mgr. Petr Páta, Ph.D.
Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt

Student's signature