CHAPTER 1
INTRODUCTION

The Rodent Activity Monitoring System (RAMS) is a specialized tool used in drug research to study the effects of drugs or substances on rodent behavior. It is designed to monitor and record the activity levels and patterns of rodents, such as mice or rats, in laboratory settings. By analyzing the behavior of rodents, researchers can gain valuable insights into the potential effects of drugs on the central nervous system and behavior.

To conduct experiments using the RAMS, rodent are usually introduced into the monitoring chamber and allowed to acclimate to the environment. The camera within the chamber detect and measure its movement and behavior. The data collected can include factors such as distance travelled, path, and rodent Area of specific behavior, and other relevant parameters.

Researchers can compare the behavior of rodents before and after drug administration to assess the drug's effects. By analyzing the data generated by the RAMS, researchers can identify changes in activity levels, alterations in circadian rhythms, abnormal patterns, or other behavioral variations caused by the drug under investigation. This information is crucial in understanding the potential therapeutic effects or side effects of the drug being studied.

The RAMS is a valuable tool in drug research as it provides quantitative and objective measurements of rodent behavior, reducing biases associated with subjective observation. It allows researchers to collect a large amount of data continuously and monitor subtle changes in behavior, providing a comprehensive understanding of the drug's impact.

Overall, the Rodent Activity Monitoring System plays a vital role in drug research, enabling scientists to assess the behavioral effects of drugs on rodents. By studying these effects, researchers can gain valuable insights into the potential efficacy and safety of drugs before further testing and development.

1.1 Problem Statement

Rodent infestations can cause significant damage to property and pose health risks to humans. Traditional methods for monitoring rodent activity, such as manual inspection

and trapping, are time-consuming and often ineffective. There is a need for a reliable and efficient system to monitor rodent activity in real-time, enabling early detection of infestations and prompt action to mitigate damage and health risks. The goal of this project is to develop a rodent activity monitoring system that utilizes modern technology to accurately detect and track rodent activity in a given area.

1.2 Objective

The objective of the mini-project is to develop a Rodent Activity Monitoring System (RAMS) which uses Object Detection to track a rodent's activity. The device aims to accurately track and record rodents, behavioural patterns and activities, typically mice or rats, within a controlled laboratory environment. The aim is to replace the current system which uses Lasers to track the rodent movement.

1.3 Scope

There is a significant scope for a rodent activity monitoring system in drug research. Rodents, particularly mice and rats, are commonly used in preclinical drug development and research to understand the safety, efficacy, and mechanisms of action of potential drug candidates. Monitoring the activity and behaviour of rodents is essential for obtaining accurate data and evaluating the effects of drugs or experimental interventions. It enables researchers to objectively assess drug efficacy, safety, pharmacokinetics, disease modelling, and phenotypic characterization, leading to more informed decision-making during the drug development process.

CHAPTER 2
LITERATURE SURVEY

2.1 Learning OpenCV

"This library is useful for practitioners, and is an excellent tool for those entering the field: it is a set of computer vision algorithms that work as advertised."-William T. Freeman, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology

Summary: "Learning OpenCV" is a comprehensive book that serves as a practical guide to understanding and utilizing the OpenCV library for computer vision applications. The book covers a wide range of topics, making it suitable for both beginners and experienced programmers interested in computer vision.

The book begins with an introduction to computer vision and provides an overview of OpenCV, its features, and its installation process. It then delves into fundamental concepts such as image manipulation, basic image processing techniques, and working with video streams. Readers learn how to handle images, apply filters and transformations, and extract various image features.

As the book progresses, it covers more advanced topics, including object detection and tracking, motion analysis, and feature extraction. It explores techniques like edge detection, image segmentation, and contour detection, enabling readers to identify and analyze objects in images and videos. Machine learning algorithms for computer vision tasks, such as classification and clustering, are also discussed.

Furthermore, "Learning OpenCV" explores 3D reconstruction and stereo vision techniques, enabling readers to create 3D models from multiple images. It also touches on augmented reality, facial recognition, and deep learning with OpenCV, providing insights into cutting-edge applications.

Throughout the book, readers are presented with practical examples and code snippets that illustrate the concepts and techniques discussed. The authors emphasize hands-on learning and provide guidance on implementing various projects using OpenCV. The code examples are explained in detail, making it easier for readers to understand and modify them according to their needs.

In summary, "Learning OpenCV" is a comprehensive and practical book that covers the essentials of computer vision using the OpenCV library. It provides a solid foundation in image processing, object detection and tracking, motion analysis, and various other computer vision techniques. With its focus on hands-on learning and real-world applications, the book equips readers with the knowledge and skills necessary to apply OpenCV effectively in their own projects.

2.1.1 Overview

OpenCV is an open source (see http://opensource.org) computer vision library available from http://SourceForge.net/projects/opencvlibrary. The library is written in C and C++ and runs under Linux, Windows and Mac OS X. Th ere is active development on interfaces for Python, Ruby, Matlab, and other languages. OpenCV was designed for computational effi ciency and with a strong focus on realtime applications. OpenCV is written in optimized C and can take advantage of multicore processors. If you desire further automatic optimization on Intel architectures , you can buy Intel's Integrated Performance Primitives (IPP) libraries, which consist of low-level optimized routines in many diff erent algorithmic areas. OpenCV automatically uses the appropriate IPP library at runtime if that library is installed. One of OpenCV's goals is to provide a simple-to-use computer vision infrastructure that helps people build fairly sophisticated vision applications quickly. The OpenCV library contains over 500 functions that span many areas in vision, including factory product inspection, medical imaging, security, user interface, camera calibration, stereo vision, and robotics. Because computer vision and machine learning oft en go hand-inhand, OpenCV also contains a full, general-purpose Machine Learning Library (MLL). Th is sublibrary is focused on statistical pattern recognition and clustering. The MLL is highly useful for the vision tasks that are at the core of OpenCV's mission, but it is general enough to be used for any machine learning problem.

2.1.1.1 What Is Computer Vision?

Computer vision is the transformation of data from a still or video camera into either a decision or a new representation. All such transformations are done for achieving some particular goal. The input data may include some contextual information such as "the camera is mounted in a car" or "laser range finder indicates an object is 1 meter

away". Th e decision might be "there is a person in this scene" or "there are 14 tumor cells on this slide". A new representation might mean turning a color image into a grayscale image or removing camera motion from an image sequence. Because we are such visual creatures, it is easy to be fooled into thinking that computer vision tasks are easy. How hard can it be to fi nd, say, a car when you are staring at it in an image? Your initial intuitions can be quite misleading. The human brain divides the vision signal into many channels that stream different kinds of information into your brain. Your brain has an attention system that identifies, in a task-dependent way, important parts of an image to examine while suppressing examination of other areas. There is massive feedback in the visual stream that is, as yet, little understood. Th ere are widespread associative inputs from muscle control sensors and all of the other senses that allow the brain to draw on cross-associations made from years of living in the world. The feedback loops in the brain go back to all stages of processing including the hardware sensors themselves (the eyes), which mechanically control lighting via the iris and tune the reception on the surface of the retina.

In a machine vision system, however, a computer receives a grid of numbers from the camera or from disk, and that's it. For the most part, there's no built-in pattern recognition, no automatic control of focus and aperture, no cross-associations with years of experience. For the most part, vision systems are still fairly naïve. Figure 2.1.1.2 shows a picture of an automobile. In that picture we see a side mirror on the driver's side of the car. What the computer "sees" is just a grid of numbers. Any given number within that grid has a rather large noise component and so by itself gives us little information, but this grid of numbers is all the computer "sees".

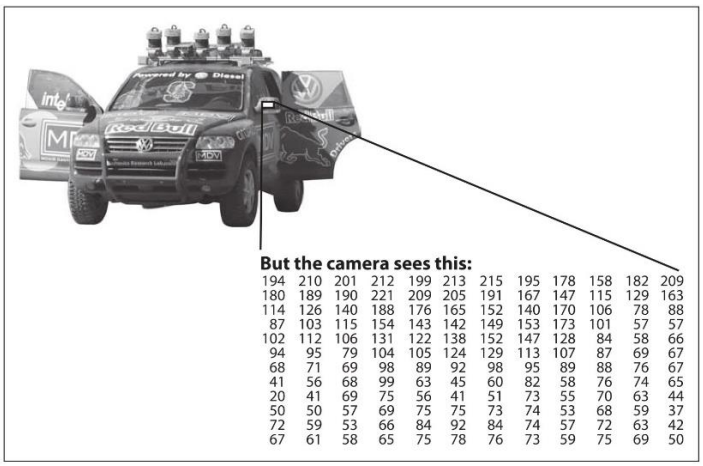


Figure 2.1.1.2 IMAGE GRID

2.2 Structured evaluation of rodent behavioral tests used in drug discovery research.

A large variety of rodent behavioral tests are currently being used to evaluate traits such as sensory-motor function, social interactions, anxiety-like and depressive-like behavior, substance dependence and various forms of cognitive function. Most behavioral tests have an inherent complexity, and their use requires consideration of several aspects such as the source of motivation in the test, the interaction between experimenter and animal, sources of variability, the sensory modality required by the animal to solve the task as well as costs and required work effort. Of particular importance is a test's validity because of its influence on the chance of successful translation of preclinical results to clinical settings. High validity may, however, have to be balanced against practical constraints and there are no behavioral tests with optimal characteristics. The design and development of new behavioral tests is therefore an ongoing effort and there are now well over one hundred tests described in the contemporary literature. Some of them are well established following extensive use, while others are novel and still unproven. The task of choosing a behavioral test for a particular project may therefore be daunting and the aim of the present review is to provide a structured way to evaluate rodent behavioral tests aimed at drug discovery research.

2.1.1 Introduction

Charles Darwin may be considered to be the founder of behavioralresearch (Thierry, 2010). Since then, behavioral testing has beenextensively used to gain a better understanding of the central nervous system (CNS) and to find treatments for its diseases. Earlyexperimental work on animal behavior includes Ivan Pavlov’swork on conditional reflexes in dogs which began at the end ofthe 19th century (Samoilov, 2007). Continued interest in animalbehavior gave rise to the field of ethology which resulted inthe Nobel Prize in Physiology and Medicine in 1973 shared by Karl von Frisch, Konrad Lorenz and Nikolaas Tinbergen. Theystudied animals in their natural habitat, which made controlledexperiments difficult. This problem was addressed by introducingbehavioral testing in a laboratory setting in the early twentiethcentury, which evolved into the field of comparative psychologyin a process facilitated by the important contributions made by BF Skinner (Gray, 1973; Dews et al.,

1994).Historically, a large variety of species has been used for behav-ioral testing but rodents have always been widely used, likely sincethey are mammals and easy to house and breed. In contrast tocommon pets such as cats and dogs, there may also be a higheracceptance in the general public for the use of rodents in medicalresearch. Although hamsters, guinea pigs and Mongolian gerbilshave been subjected to behavioral testing, mice and rats are farmore popular and firmly established as model organisms withseveral outbred stocks and inbred strains available for experi-ments. Unlike other rodents used for research, mice and ratsbelong to the subfamily Murinaea and are sometimes referred toas murine models. Early examples of rodent behavioral testinginclude Karl Lashley's work on learning and memory using mazesin the early twentieth century. Initially, wild-caught rodents wereused for experiments (McCoy, 1909) but this practice changedwith the introduction of strains bred by mouse and rat fanciers(Steensma et al., 2010). Following approximately a hundred yearsof breeding, contemporary laboratory animals are now consid-erably more docile than their wild counterparts (Wahlsten et al.,2003a). Over time, there has been a continuous evolution ofrodent behavioral tests and there are well over 100 tests in contem-porary use, exhaustively summarized in Supplementary Table of the present review. In recent years, genetically modified mice havebecome readily available and the use of mice in behavioral testingrecently surpassed that of rats (Figure 1). However, the increasingavailability of genetically modified rats (Jacob et al., 2010) mayshift the tide again.

CHAPTER – 3
PROPOSED WORK

The objective of this project (RAMS) is to develop and evaluate an automated rodent activity monitoring system designed to enhance drug research studies. The system leverages advanced technologies, including computer vision, machine learning, and sensor integration, to monitor and analyze the behavioral patterns of rodents in a controlled laboratory environment. The primary objective of this work is to enhance the accuracy, reliability, and efficiency of drug research by providing a comprehensive and automated analysis of rodent behavior.

3.1. Methodology

1. System Design: Design and develop a specialized enclosure equipped with high-resolution cameras, motion sensors, and environmental sensors for precise data collection and environmental control. Ensure the system is scalable and adaptable to different experimental setups.

2. Data Collection: Conduct experiments using rodent models and administer pharmaceutical compounds of interest. Capture high-quality video recordings of the rodents' behavior within the specially designed enclosure while monitoring environmental variables.

3. Data Processing: Develop computer vision algorithms to track and extract relevant features from the recorded videos, such as locomotion, exploratory behavior, and social interactions. Implement machine learning algorithms to identify behavioral patterns and establish correlations between drug administration and specific behavioral changes.

4. System Validation: Evaluate the accuracy and reliability of the automated rodent activity monitoring system by comparing its results with manual observations and existing gold standard methods. Assess the system's ability to detect subtle behavioral changes induced by drug administration accurately.

5. System Performance and Efficiency: Measure the efficiency and processing speed of the automated system, comparing it to manual data analysis methods. Determine the system's ability to handle large datasets and extract meaningful insights efficiently.

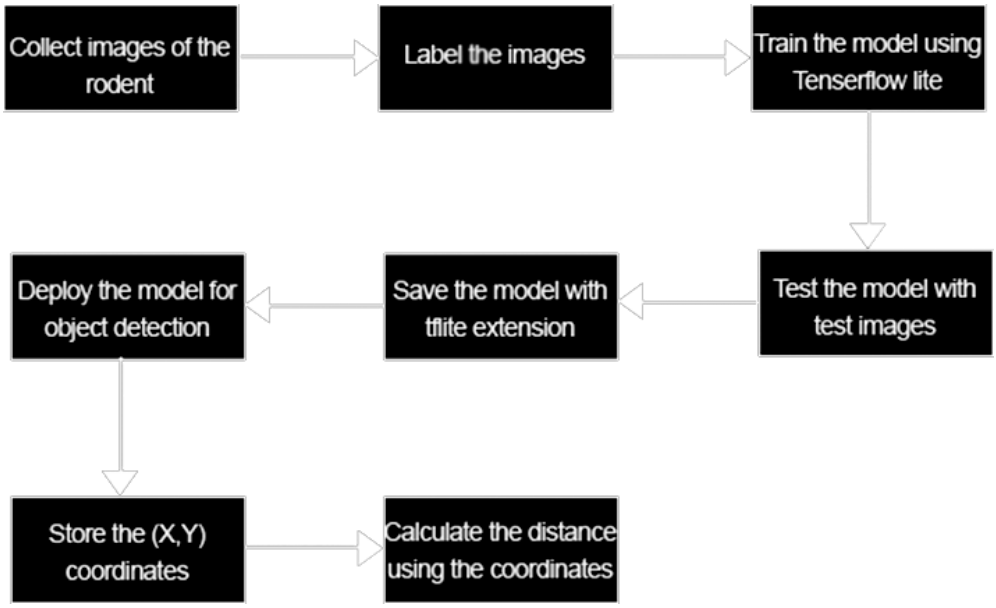
6. Practical Application: Demonstrate the system's practical applicability by conducting case studies with different pharmaceutical compounds and evaluating their behavioral effects. Assess the system's effectiveness in identifying potential drug candidates or optimizing existing therapeutic compounds.

3.2 Expected Outcomes:

1. A robust and scalable automated rodent activity monitoring system for drug research studies.
2. Validation of the system's accuracy and reliability compared to manual observations.
3. Demonstration of the system's ability to detect subtle behavioral change induced by drug administration.
4. Improved efficiency in data processing and analysis, leading to faster extraction of meaningful insights.
5. Case studies showcasing the practical application of the system in identifying potential drug candidates or optimizing therapeutic compounds.

This proposed work aims to contribute to the advancement of drug research by providing a sophisticated and automated system for monitoring and analyzing rodent behavior. The outcomes of this research can enhance the understanding of drug effects on behavior and support the development of novel therapeutic interventions.

3.3. PROCESS FLOW DIAGRAM



1. Collect images of the rodent

Collecting images of rodents refers to the process of capturing visual data of rodents through the use of cameras or imaging devices. In the context of developing a rodent activity monitoring system for drug research, collecting images involves capturing photographs or video recordings of rodents in a controlled laboratory environment.

2. Label the images

Labelling the images involves the process of drawing bounding boxes around the rodents and defining the class to indicate their presence and location within the frame. This bounding box annotation provides spatial information about the rodents' position and facilitates subsequent analysis and tracking.

3. Train the model using tensorflow lite

Training a model using TensorFlow Lite involves the process of developing a machine learning model using the TensorFlow framework and then converting it to the TensorFlow Lite format for deployment on resource-constrained devices. TensorFlow Lite enables the deployment of the model on resource-constrained

devices, making it suitable for integration into the rodent activity monitoring system for drug research.

4. Test the model with the test images

Testing the model with test images involves evaluating the performance of the trained TensorFlow Lite model on a separate dataset that was not used during the training phase. Testing the TensorFlow Lite model with test images provides a crucial step in assessing the model's accuracy and robustness in detecting and localizing rodents. By evaluating the model's performance on an independent dataset, you can gain insights into its generalization capabilities and identify areas for further refinement or optimization.

5. Save the model with tflite extension

Since detection of the rodent is done in tensorflow lite, the model have to be saved in tflite extension. By saving the model in the TensorFlow Lite format, you ensure that it is optimized for efficient execution on mobile and embedded platforms while preserving its functionality and reducing its size.

6. Deploy the model for object detection function

Deploying a TensorFlow Lite model for object detection involves integrating the model into an application or system that can take input images or video frames and perform real-time object detection using the deployed model.

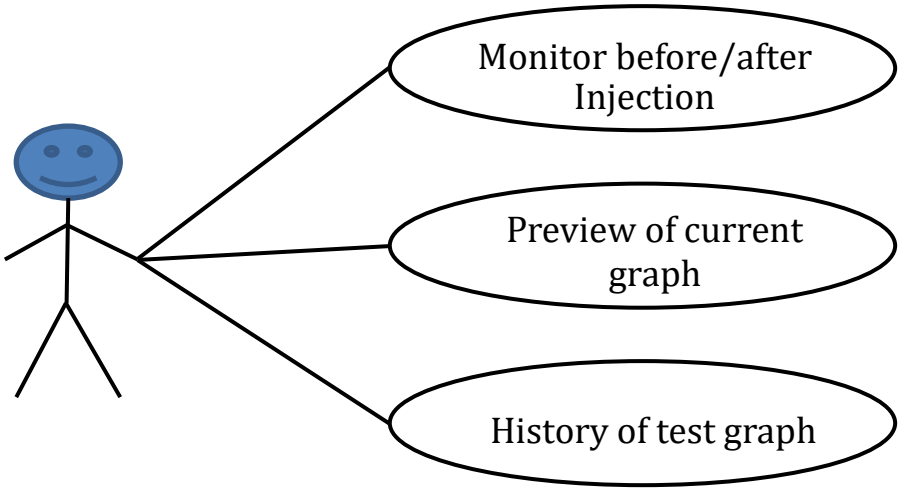
7. Store X,Y coordinates

To find the distance travelled by the rodent, the (x,y) coordinates of the center the bounding box have to be saved.

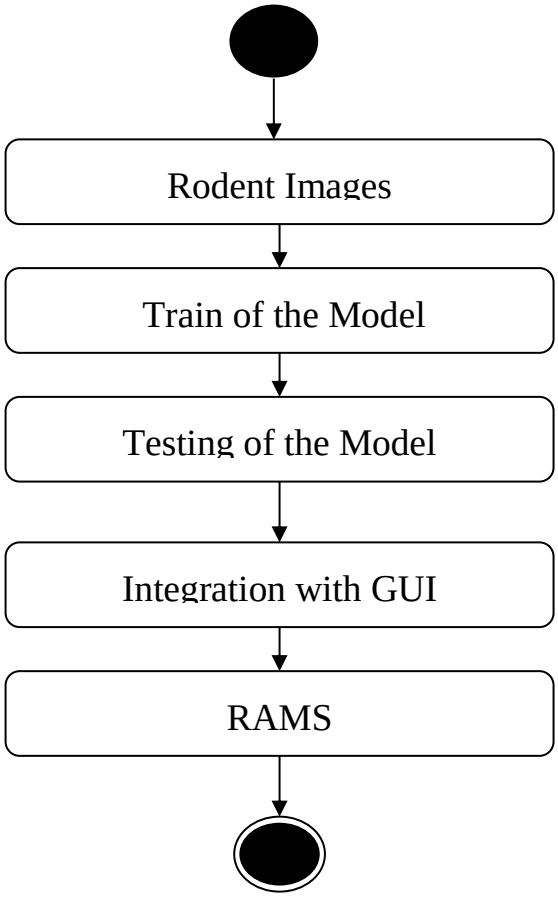
8. Calculate the distance using coordinates

To calculate the distance between two points using their coordinates (x1, y1) and (x2, y2), you can apply the Euclidean distance formula. The Euclidean distance is the straight-line distance between two points in a two-dimensional space.

3.4 Use Case Diagram



3.5 Activity Diagram



CHAPTER 4

Results and Discussion

Rodent activity monitors based on image detection using TensorFlow Lite using machine learning frameworks can be valuable tools in scientific research and pest control. These monitors typically involve the use of cameras or other imaging devices to capture video footage of rodent behavior. The recorded video is then processed using machine learning algorithms to detect and track rodent activity.

The implementation of TensorFlow Lite in this context allows for real-time processing of the captured video directly on resource-constrained devices such as embedded systems, smartphones, or microcontrollers. TensorFlow Lite is a lightweight version of the TensorFlow framework specifically optimized for deployment on edge devices.

The result and discussion of a rodent activity monitor using TensorFlow Lite would depend on several factors, including the accuracy and performance of the image detection model, the quality of the training data, the complexity of the rodent behaviors being monitored, and the specific goals of the study or application.

The evaluation of such a system would typically involve comparing the detected rodent activity with ground truth data obtained through manual observation or other established methods. Metrics such as precision, recall, and F1 score may be used to assess the accuracy of the detection model. The system's ability to detect various types of rodent behaviors, such as locomotion, grooming, or feeding, could also be evaluated.

By this method researchers would analyze the strengths and limitations of the system, discussing its performance, potential sources of error and possible areas for improvement. They may also compare their results with existing rodent monitoring techniques to highlight the advantages and disadvantages of using image detection with TensorFlow Lite.

It's important to note that the specific results and discussions would vary depending on the study's objectives, the experimental setup, and the data collected.

CHAPTER 5

CONCLUSION AND FUTURE WORK

The Rodent Activity Monitoring System (RAMS) is an invaluable tool in drug research, facilitating the study of drug effects on rodent behavior. By providing objective and quantitative measurements, RAMS reduces biases associated with subjective observation and allows researchers to monitor subtle changes in behavior. The system's ability to continuously collect data, analyze behavior, and compare pre- and post-drug administration results provides comprehensive insights into the drug's impact on the central nervous system and behavior. RAMS plays a pivotal role in preclinical research, aiding in the identification of potential drug candidates with promising behavioral effects for further investigation.

Despite its significant contributions to drug research, the Rodent Activity Monitoring System offers opportunities for further improvement and expansion:

1. Integration of advanced behavioral metrics: Future versions of RAMS could incorporate more sophisticated behavioral metrics to provide a more detailed analysis of rodent behavior. This could include parameters like social interactions, cognitive tasks, and complex movement patterns.

2. Multi-modal data collection: Integrating other types of data, such as physiological measurements (heart rate, temperature, etc.) and neurobiological data (brain activity), could enhance researchers' understanding of the drug's effects on both behavior and physiological responses.

3. Automated data analysis and machine learning: Developing automated data analysis algorithms and incorporating machine learning techniques could streamline the processing of large datasets generated by RAMS. This would help researchers uncover hidden patterns and relationships within the data more efficiently.

4. Application in disease models: Expanding the use of RAMS to study drug effects in specific disease models, such as neurodegenerative disorders or psychiatric conditions, could provide valuable insights into potential therapeutic interventions.

5. Long-term monitoring: Enhancing RAMS to enable extended long-term monitoring of rodent behavior would allow researchers to observe behavioral changes over more extended periods, potentially revealing progressive effects of drugs.

6. Translation to human studies: Exploring the feasibility of adapting similar monitoring systems to study human behavior and drug effects in controlled settings could bridge the gap between preclinical and clinical research.

7. Collaborative data sharing: Establishing platforms for researchers to share RAMS data and collaborate on large-scale studies could facilitate data-driven discoveries and foster a more comprehensive understanding of drug effects across various contexts.

REFERENCES

[1] Learning OpenCV [Book] (2008)[accessed Feb. 25 2023]

[2] Structured evaluation of rodent behavioral tests used in drug discovery research. (2014)[accessed Feb. 25 2023]

Appendix A SAMPLE CODE

```
from PyQt5.uic import loadUi
from PyQt5 import QtCore, QtGui, QtWidgets
from PyQt5.QtGui import QPixmap,QImage,QColor
from PyQt5.QtCore import pyqtSignal, pyqtSlot, Qt, QThread,QAbstractListModel
import time
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
import sys
import cv2
import detector
import CSV_reader


inj=False
quit_th=False
qImg=QImage()
resultant_list=[]
text=""

class VideoThread(QThread):
    #signal = VideoSignal()
    change_pixmap_signal = pyqtSignal(QImage)
    finished_running_signal = pyqtSignal(str)
    def __init__(self, inj1,path=2, curFrame=0):
        global inj
        super(VideoThread, self).__init__()
        self.cap = cv2.VideoCapture(2)
        # self.signal = VideoSignal()
        self.curFrame = curFrame
```

```
        inj=inj1
        self.is_killed = False

    def run(self):
        global resultant_list,inj
        self.start_time = time.monotonic()
        self.is_killed = False
        j = self.curFrame
        self.cap.set(2, j)
        while (self.cap.isOpened() and not self.is_killed):
            rep, frame = self.cap.read()
            self.curFrame = j
            if not rep:
                break
            height, width, channel = frame.shape
            frame=detector.run(frame,inj,text)
            bytesPerLine = 3 * width
            QImg = QImage(frame.data, width, height,
bytesPerLine,QImage.Format_BGR888)
            QImg = QImg.scaled(908, 681, Qt.KeepAspectRatio)
            self.change_pixmap_signal.emit(QImg)
            j += 1
            if(time.monotonic()-self.start_time>=120):
                detector.save_csv()
                resultant_list=detector.save_csv_his()
                self.finished_running_signal.emit("Finished...")
                self.is_killed=True
                self.quit()
                self.cap.release()


    def stop(self):
```

```
        self.is_killed = True
        self.quit()
        self.cap.release()



class CountThread(QThread):
    change_count_signal = pyqtSignal(str)

    def __init__(self,count=120):
        super(CountThread, self).__init__()
        self.is_kill = False
        self.count = count

    def run(self):
        self.is_kill = False
        while (self.count != 0 and not self.is_kill):
            min = int(self.count / 60)
            sec = int((self.count % 60))
            if sec < 10:
                temp="0"+str(sec)
            else:
                temp=str(sec)
            if min<10:
                tem="0"+str(min)
            else:
                tem=str(min)
            tm=tem+"M "+temp+"S"
            self.change_count_signal.emit(tm)
            if self.count<=0:
                self.is_kill=True
                self.quit()
```

```
            self.count=self.count-1
            time.sleep(1)

    def stop(self):
        self.is_kill = True
        self.quit()

class WelcomeScreen(QMainWindow):
    def __init__(self):
        super(WelcomeScreen, self).__init__()
        loadUi("RAM_MAIN.ui",self)
        font_path = "Assest/arlrdbd.ttf"

        QFontDatabase.addApplicationFont(font_path)
        font = QFont("font_path",114)
        self.label.setFont(font)
        self.Scan_btn.clicked.connect(self.scan_ro)
        self.hist_btn.clicked.connect(self.Hist_ro)
        self.quit_btn.clicked.connect(self.exit_sc)


    def scan_ro(self):
        deci=DecisionScreen()
        widget.addWidget(deci)
        widget.setCurrentWidget(deci)

    def Hist_ro(self):
        hist=HistoryScreen()
        widget.addWidget(hist)
        widget.setCurrentWidget(hist)
```

```
    def exit_sc(self):
        widget.close()

    def mousePressEvent(self, event):
        self.oldPosition = event.globalPos()

    def mouseMoveEvent(self, event):
        delta = QPoint(event.globalPos() - self.oldPosition)
        widget.move(widget.x() + delta.x(), widget.y() + delta.y())
        self.oldPosition = event.globalPos()

class DecisionScreen(QMainWindow):

    def __init__(self):

        super(DecisionScreen, self).__init__()
        loadUi("dec.ui",self)
        self.after_btn.clicked.connect(self.moni_ro_1)
        self.before_btn.clicked.connect(self.moni_ro)
        self.back_btn.clicked.connect(self.goback)

    def moni_ro_1(self):
        global text,inj
        inj=True
        monit = MonitorScreen()
        widget.addWidget(monit)
        widget.addWidget(monit)
        widget.setCurrentIndex(widget.currentIndex() + 1)
        text = self.in_tx.toPlainText()
    def moni_ro(self):
        global text,inj
        inj = False
```

```
        monit = MonitorScreen()
        widget.addWidget(monit)
        widget.setCurrentWidget(monit)
        text = self.in_tx.toPlainText()
    def goback(self):
        ba=WelcomeScreen()
        widget.addWidget(ba)
        widget.setCurrentWidget(ba)

    def mousePressEvent(self, event):
        self.oldPosition = event.globalPos()

    def mouseMoveEvent(self, event):
        delta = QPoint(event.globalPos() - self.oldPosition)
        widget.move(widget.x() + delta.x(), widget.y() + delta.y())
        self.oldPosition = event.globalPos()

class ResultScreen(QMainWindow):

    def __init__(self):
        super(ResultScreen, self).__init__()
        loadUi("Result.ui",self)
        self.back_btn.clicked.connect(self.goback)
        self.Test_name.setText(resultant_list[0])
        self.Distance_be.setText(str(resultant_list[2]))
        self.Distance_af.setText(str(resultant_list[1]))
        self.Date_of_test.setText(str(resultant_list[3]))
        self.Area_of_rodent.setText(str(resultant_list[4]))
        self.excersity.setText(str(resultant_list[5]))
        self.set_image()
        QtCore.QTimer.singleShot(3, lambda: self.set_image())
```

```
    def set_image(self):
        image_path="PLOTS/"+resultant_list[0]+".png"
        image_path_dvt="PLOTS/DVT_"+resultant_list[0]+".png"
        image_path_avt="PLOTS/AVT_"+resultant_list[0]+".png"

        pixmap = QPixmap(image_path)
        pixmap_dvt = QPixmap(image_path_dvt)
        pixmap_avt = QPixmap(image_path_avt)

        pixmap_re=pixmap.scaled(511,311)
        pixmap_dvt_re=pixmap_dvt.scaled(511,311)
        pixmap_avt_re=pixmap_avt.scaled(511,311)

        self.Graph_label_path.setPixmap(pixmap_re)
        self.Graph_label_dvt.setPixmap(pixmap_dvt_re)
        self.Graph_label_avt.setPixmap(pixmap_avt_re)
    def goback(self):
        ba=WelcomeScreen()
        widget.addWidget(ba)
        widget.setCurrentWidget(ba)

    def mousePressEvent(self, event):
        self.oldPosition = event.globalPos()

    def mouseMoveEvent(self, event):
        delta = QPoint(event.globalPos() - self.oldPosition)
        widget.move(widget.x() + delta.x(), widget.y() + delta.y())
        self.oldPosition = event.globalPos()

class MonitorScreen(QMainWindow):
    def __init__(self):
        global inj
```

```
        super(MonitorScreen, self).__init__()
        loadUi("scan.ui",self)
        self.str.clicked.connect(self.play)
        self.back_btn.clicked.connect(self.goback)
        self.grey = QPixmap(908, 681)
        self.grey.fill(QColor('darkGray'))
        self.label_3.setPixmap(self.grey)
        self.label_progress.setText("")
        self.count_time.setText("00M 00S")
        self.cj=1
    def ImageUpdateSlot(self, Image):
        self.label_3.setPixmap(QPixmap.fromImage(Image))
    def TimeUpdateSlot(self,tm):
        self.count_time.setText(tm)
    def Finished_update(self,ran):
        global inj
        self.label_progress.setText(ran)
        if (inj==True):
            print(inj)
            res = ResultScreen()
            widget.addWidget(res)
            widget.setCurrentWidget(res)

        else:
            print(inj)
            ba = WelcomeScreen()
            widget.addWidget(ba)
            widget.setCurrentWidget(ba)
    def play(self):
        global inj
        self.cj=0
        self.label_progress.setText("Processing..")
```

```
        self.worker1 = VideoThread(inj,path=0)
        self.worker2=CountThread()
        self.worker1.start()
        self.worker2.start()
        self.worker1.change_pixmap_signal.connect(self.ImageUpdateSlot)
        self.worker1.finished_running_signal.connect(self.Finished_update)
        self.worker2.change_count_signal.connect(self.TimeUpdateSlot)
        self.back_btn.clicked.connect(self.goback)
        self.grey = QPixmap(908, 681)
        self.grey.fill(QColor('darkGray'))
        self.label_3.setPixmap(self.grey)
    def goback(self):
        self.label_progress.setText("")
        if(self.cj==0):
            self.worker1.stop()
            self.worker2.stop()
        self.cj=1
        ba=WelcomeScreen()
        widget.addWidget(ba)
        widget.setCurrentWidget(ba)

    def mousePressEvent(self, event):
        self.oldPosition = event.globalPos()

    def mouseMoveEvent(self, event):
        delta = QPoint(event.globalPos() - self.oldPosition)
        widget.move(widget.x() + delta.x(), widget.y() + delta.y())
        self.oldPosition = event.globalPos()
class WidgetListModel(QAbstractListModel):
    def __init__(self, data=None):
        super().__init__()
        self.data = data or []
```

```
    def rowCount(self, parent=QModelIndex()):
        return len(self.data)
    def data(self, index, role=Qt.DisplayRole):
        if role == Qt.DisplayRole:
            widget = self.data[index.row()]
            if isinstance(widget, QLabel):
                return widget.text()
        if role == Qt.FontRole:
            font = QFont()
            font.setPointSize(20)  # Set the desired font size
            return font

        return None
class WidgetDelegate(QStyledItemDelegate):
    def paint(self, painter, option, index):
        #option.displayAlignment = Qt.AlignCenter
        super().paint(painter, option, index)
        # Modify the font size
        font = painter.font()
        font.setPointSize(26)  # Set the desired font size
        painter.setFont(font)

    def sizeHint(self, option, index):
        size = super().sizeHint(option, index)
        size.setHeight(45)  # Set the desired height for each item
        return size
class HistoryScreen(QMainWindow):
    def __init__(self):
        super(HistoryScreen, self).__init__()
        loadUi("history.ui",self)
        self.back_btn_hist.clicked.connect(self.goback)
        count=CSV_reader.count_csv_rows("History.csv")
```

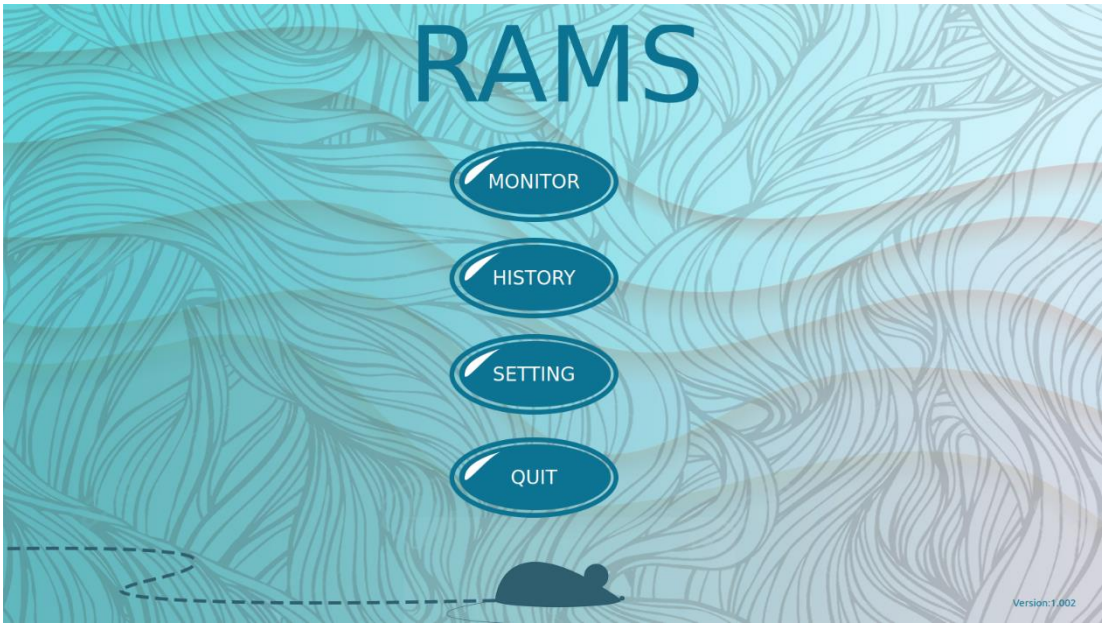
```
wid=[]
if(count<10):
    for i in range(count):
        wn="widget"+str(i)
        wid.append(QLabel(wn))
    dat=CSV_reader.get_data("History.csv")
    j=len(dat)-1
    k=0
    while(j>=0):
        tem=dat[j]
        wid[k].setText(str(k+1)+".      "+str(tem[0])+"  ("+str(tem[3]+")"))
        k=k+1
        j=j-1
else:
    for i in range(10):
        wn="widget"+str(i)
        wid.append(QLabel(wn))
    dat=CSV_reader.get_data("History.csv")
    j=len(dat)-1
    k=0
    while(j>=0):
        tem=dat[j]
        wid[k].setText(str(k+1)+".      "+str(tem[0])+"  ("+str(tem[3]+")"))
        k=k+1
        j=j-1
model = WidgetListModel(wid)

self.listView.setModel(model)
delegate = WidgetDelegate()
self.listView.setItemDelegate(delegate)
self.listView.clicked.connect(self.navigate)
```

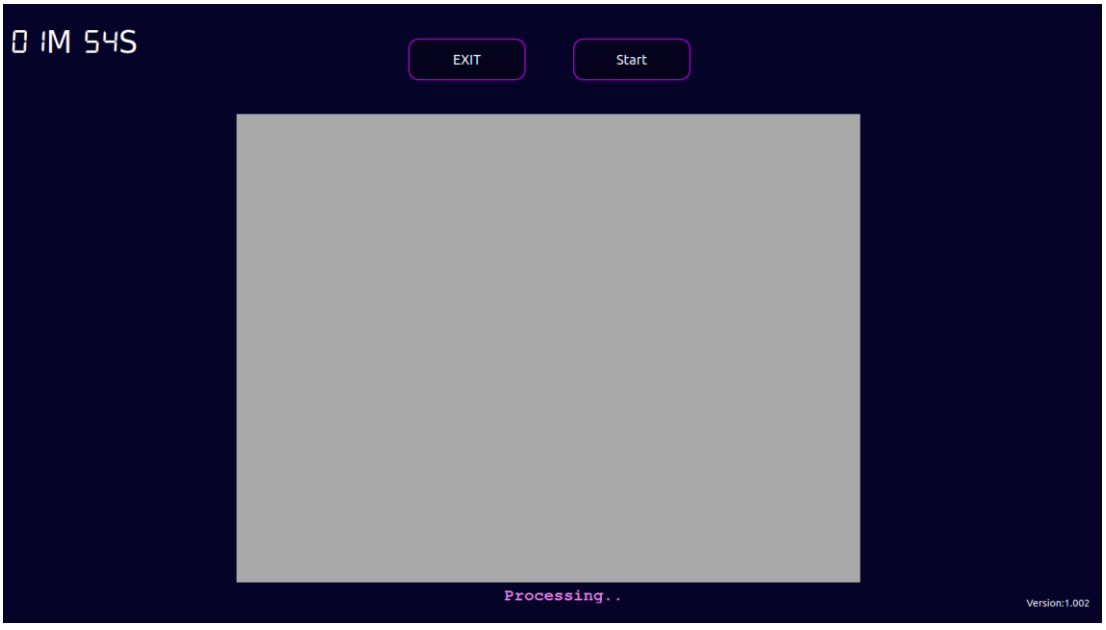
```
    def navigate(self, index):
        # Get the selected item's data
        widget1 = index.data(Qt.DisplayRole)
        word_to_remove=str(widget1).split()[0]
        new_string = str(widget1).replace(word_to_remove, "")
        words = new_string.split()
        new_string = ' '.join(words[:-1])
        new_string = new_string.strip()
        res=ResScreen(new_string)
        widget.addWidget(res)
        widget.setCurrentWidget(res)

    def goback(self):
        ba=WelcomeScreen()
        widget.addWidget(ba)
        widget.setCurrentWidget(ba)

    def mousePressEvent(self, event):
        self.oldPosition = event.globalPos()

    def mouseMoveEvent(self, event):
        delta = QPoint(event.globalPos() - self.oldPosition)
        widget.move(widget.x() + delta.x(), widget.y() + delta.y())
        self.oldPosition = event.globalPos()

class ResScreen(QMainWindow):
    def __init__(self,NAME):
        super(ResScreen, self).__init__()
        loadUi("Res_his.ui",self)
        self.Test_name.setText(NAME)
        self.back_btn.clicked.connect(self.goback)
```

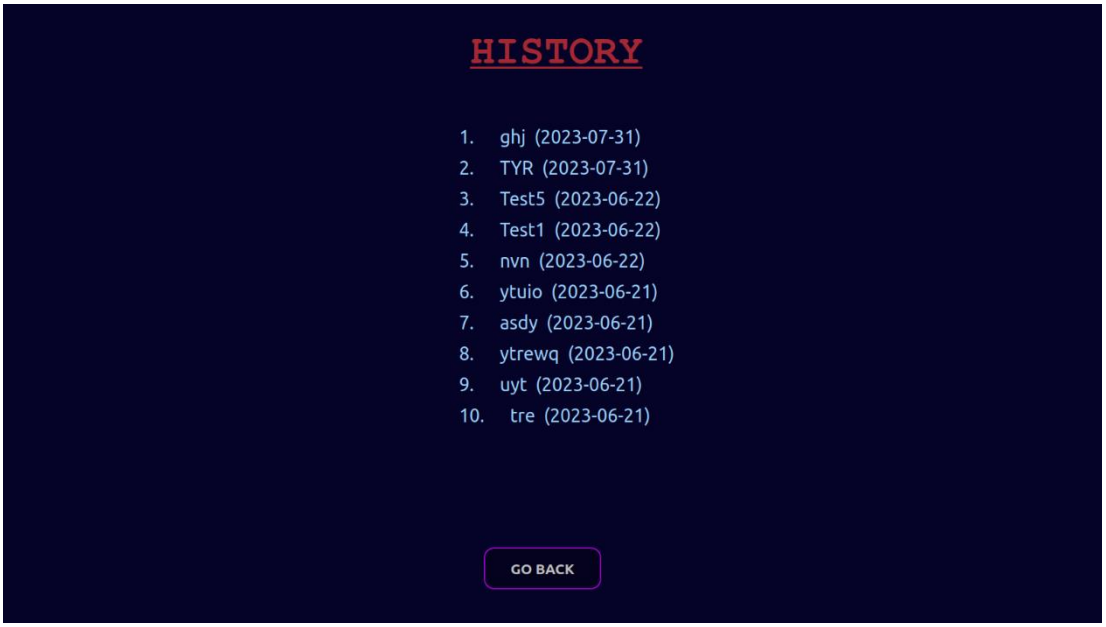
```
        dat = CSV_reader.get_data("History.csv")
        for row in dat:
            if NAME in row:
                self.Distance_af.setText(str(row[1]))
                self.Distance_be.setText(str(row[2]))
                self.Date_of_test.setText(str(row[3]))
                self.Area_of_rodent.setText(str(row[4]))
                self.excersity.setText(str(row[5]))
        self.set_image(row[0])
        QtCore.QTimer.singleShot(3, lambda: self.set_image(row[0]))

    def set_image(self,NAME):
        image_path = "PLOTS/" + NAME + ".png"
        image_path_dvt = "PLOTS/DVT_" + NAME + ".png"
        image_path_avt = "PLOTS/AVT_" + NAME + ".png"

        pixmap = QPixmap(image_path)
        pixmap_dvt = QPixmap(image_path_dvt)
        pixmap_avt = QPixmap(image_path_avt)

        pixmap_re = pixmap.scaled(511, 311)
        pixmap_dvt_re = pixmap_dvt.scaled(511, 311)
        pixmap_avt_re = pixmap_avt.scaled(511, 311)

        self.Graph_label_path.setPixmap(pixmap_re)
        self.Graph_label_dvt.setPixmap(pixmap_dvt_re)
        self.Graph_label_avt.setPixmap(pixmap_avt_re)

    def goback(self):
        ba=HistoryScreen()
        widget.addWidget(ba)
        widget.setCurrentWidget(ba)
```

```
    def mousePressEvent(self, event):
        self.oldPosition = event.globalPos()

    def mouseMoveEvent(self, event):
        delta = QPoint(event.globalPos() - self.oldPosition)
        widget.move(widget.x() + delta.x(), widget.y() + delta.y())
        self.oldPosition = event.globalPos()

# main
app = QApplication(sys.argv)
welcome = WelcomeScreen()
widget = QtWidgets.QStackedWidget()
widget.setWindowFlags(QtCore.Qt.FramelessWindowHint)
widget.move(175,110)
widget.addWidget(welcome)
widget.setFixedSize(1600, 900)
widget.show()
try:
    sys.exit(app.exec_())
except:
    print("Exiting")
```

Appendix B SCREENSHOTS






HISTORY
1. ghj (2023-07-31)
2. TYR (2023-07-31)
3. Test5 (2023-06-22)
4. Test1 (2023-06-22)
5. nvn (2023-06-22)
6. ytuio (2023-06-21)
7. asdy (2023-06-21)
8. ytrewq (2023-06-21)
9. uyt (2023-06-21)
10. tre (2023-06-21)
GO BACK


RESULT
PATH
DVT
AVT
Path Graph
Distance VS Time
Area VS Time
Test Name : Test1
Distance : 2907
(Before injecting)
Distance : 1029
(After injecting)
Date of Test: 2023-06-22
Excersity: 0
Avg Area of Rodent: 25200
GO BACK