DABA: An Open-Source Platform for Streamlined Behavioral Analysis Using DeepLabCut and Customizable Local LLM-Generated Code

Farhan Augustine
University of Maryland, Baltimore County, MD.

Abstract

Understanding animal behavior is critical for advancing various scientific fields, including neuroscience, ecology, and evolutionary biology. Traditional methods for analyzing animal behavior are often labor-intensive and subjective. While pose-tracking tools like DeepLabCut (DLC) and SLEAP automate the extraction of pose-estimation, translating this data into meaningful behavioral insights remains challenging. To address this gap, I introduce DABA (Dynamic Animal Behavior Analysis), an open-source software that empowers researchers with a powerful and flexible toolkit for behavioral analysis. DABA uniquely combines a set of pre-built modules for standard analyses (such as spatial analysis, movement analysis, and event duration) with a dynamic code generation system powered by a local Large Language Model (LLM). This dual approach allows researchers to quickly quantify common behavioral metrics while also providing the flexibility to create custom analyses tailored to specific research questions, even without coding expertise. By leveraging local processing with an LLM, DABA ensures data privacy and security while enabling users to customize analyses. Adaptation of DABA by the community and its customizability have the potential to significantly streamline and enhance the analysis of animal behaviors. Source Code is hosted on GitHub accessible via: (https://github.com/farhanaugustine/DABA-Dynamic_Animal_Behavior_Analysis)

Introduction

The study of animal behavior is fundamental to scientific discovery, offering crucial insights into brain function, ecological dynamics, and evolutionary processes. However, extracting meaningful knowledge from animal movement has long posed a significant challenge. Traditional methods, such as manual video annotation, are not only labor-intensive and time-consuming but also susceptible to observer bias, limiting the scale and scope of behavioral research. While the advent of advanced pose-tracking technologies like DeepLabCut (DLC) (Mathis, et al., 2018), Social LEAP Estimates Animal Poses (SLEAP) (Pereira, et al., 2022), and many other deep learning methods, including Argos (Ray & Stopfer, 2021), FastTrack (Gallois & Candelier, 2021), and SimBA (Nilsson, et al., 2024), just to name a few, have automated the extraction of kinematic data, a critical gap remains in translating this raw data into comprehensive behavioral understanding. Many existing analysis solutions are inflexible and often require programming expertise to adapt to specific research questions, creating a barrier for researchers without a strong coding background. Furthermore, while solutions such as the recently introduced AmadeusGPT (Ye, Lauer, Zhou, Mathis, & Mathis, 2023) have attempted to simplify custom analyses for researchers, they rely on cloud-based processing via external APIs, raising substantial concerns about the privacy and security of sensitive research data. To address these challenges and empower researchers with greater flexibility and control over their behavioral analyses, I introduce DABA (Dynamic Animal Behavior Analysis), a novel open-source software designed to empower researchers with a powerful and flexible toolkit for behavioral analysis. DABA uniquely combines a suite of pre-built modules for standard analyses with a dynamic code-generation system powered by a local Large Language Model (LLM) through integration with Ollama. This dual approach enables rapid quantification of common behavioral metrics while simultaneously providing the flexibility to create custom analyses tailored to specific experimental needs, even for those without prior programming experience. By leveraging the power of large language models locally, DABA allows researchers to simply describe their desired analysis in natural language, like AmadeusGPT, and the local LLM will automatically generate the corresponding code. By prioritizing local processing, DABA ensures that valuable research data remains secure and under your control, fostering both data integrity and user autonomy. DABA's innovative approach to leverage Ollama offers the capability to streamline and enhance the analysis of animal behaviors. By providing tools for both rapid quantification of standard metrics and the development of custom analyses, DABA provides researchers with an easy-to-use tool, facilitating the acceleration of scientific discoveries and opening new avenues for understanding the complex interplay between movement, behavior, and the underlying biological mechanisms.

Software and Libraries

1. Core Libraries:

I built DABA using Python 3.10 and leveraged a collection of key libraries to enable its functionalities:

i. pandas (v2.1.4): Provides the foundational data structures for DABA, enabling efficient handling and manipulation of the tabular data outputted by DeepLabCut. It allows for the loading, cleaning, and transformation of kinematic data into a format suitable for analysis.

ii. numpy (v1.26.3): Underpins the numerical and mathematical computations within DABA. It is used for calculating various metrics such as distance, velocity, and other statistical measures derived from the positional data of tracked body parts.

iii. httpx (v0.26.0) and asyncio: These libraries work in tandem to manage asynchronous HTTP requests for efficient, non-blocking communication (i.e., the software remains responsive while waiting for the LLM). This is crucial for DABA's interaction with the local Ollama API, facilitating efficient communication with the LLM for dynamic code generation without blocking the main program flow or timing out during large responses.

iv. cv2 (v4.9.0): OpenCV (cv2) is utilized for all video processing tasks. It enables the definition of Regions of Interest (ROIs) through a graphical interface, handles video frame extraction, and supports the visualization of tracked body part paths overlaid on the video.

v. tabulate (v0.9.0): Improves the readability of analysis results by generating neatly formatted tables. This library is used to present the output of various analysis modules in a clear and organized manner within the terminal.

vi. logging: Implements a flexible event logging system that tracks the progress of analyses, records any errors or warnings, and provides a detailed history of operations performed during a session. This is essential for debugging and monitoring the software's performance.

vii. importlib: Facilitates the dynamic loading of Python modules at runtime. This capability is used to load the standard analysis modules included with DABA, allowing users to select and execute specific analyses as needed.

2. Ollama and Customizable LLMs:

DABA's innovative code generation feature is powered by a local integration with Ollama (https://ollama.com/), a platform designed for running large language models (LLMs) locally. This integration allows researchers to leverage the power of LLMs for generating custom analysis code while ensuring data privacy and security, as all processing occurs on user's machine.

To streamline the user experience, I've made two pre-configured LLMs named llama3.3_DLC available through Ollama.com. These models have been specifically tailored to understand the structure and content of DeepLabCut output data, providing a robust starting point for a wide range of behavioral analyses, offering a balance between model size, speed, and accuracy. Researchers can download and run these optimized models with simple commands: ollama run FarhanAugustine/llama3.3_DLC and ollama run FarhanAugustine/llama3.3_DLC_q3_K_S. These models are based on the llama3:70b (43GB) base model, with quantization levels of q4_K_M (43 GB) and q3_K_S (31 GB), respectively. The smaller q3_K_S model offers a reduced memory footprint, while the q4_K_M model may provide slightly better performance and accuracy.

For researchers seeking greater control and customization, DABA supports the integration of other LLMs via the `config.yaml` file. This feature allows you to develop and deploy specialized models fine-tuned to specific research needs, experimental paradigms, or computational resources. To guide researchers in creating prompts for custom models, I have provided detailed prompt templates within the DABA framework (Figure 1; Table 1). These templates help structure prompts that effectively communicate analysis requirements to the LLMs. This capability not only enhances the flexibility of DABA but also empowers researchers to push the boundaries of behavioral analysis beyond the limitations of pre-defined modules and directly incorporate LLMs within their behavioral analysis workflow.

3. Input Data and Preprocessing

DABA requires two primary inputs to initiate analysis (Figure 2):

i. DeepLabCut CSV file: This file contains the output from DeepLabCut, providing the x and y coordinates, along with their associated likelihoods, for each tracked body part across all frames of the video (i.e., the default CSV output generated by DLC). The CSV file should be formatted with columns representing frame number, body part labels, x-coordinates, y-coordinates, and likelihood scores.

ii. Corresponding video file: This video file is used for several purposes, including the definition of Regions of Interest (ROIs) (Figure 3), visualization of path tracing, and extraction of the video's frame rate for accurate time-based calculations.

Upon input, DABA loads the raw DeepLabCut data into a pandas DataFrame. Subsequently, it is reorganized into a nested dictionary for efficient processing. The dictionary uses frame numbers as primary keys, with each entry containing another dictionary mapping body part labels to their respective (x, y) coordinates and likelihood scores. This structure allows for quick access to positional data for any given body part at any given frame. All data handling and preprocessing steps are performed locally, ensuring data security and user privacy.

4. Analysis Capabilities

DABA offers a versatile suite of analysis capabilities to accommodate a wide range of research needs. These capabilities are divided into pre-defined Standard Analysis Modules for common behavioral metrics and a unique Dynamic Code Generation feature for bespoke analyses.

I. Standard Analysis Modules

DABA includes a comprehensive library of pre-defined modules designed to quantify common aspects of animal behavior. These modules provide researchers with readily accessible tools to analyze movement patterns, spatial preferences, and interactions within defined areas. Key modules include:

i. *Distance and Time in ROI:* This module calculates the total distance a specified body part travels within a user-defined Region of Interest (ROI) and the total time spent within that ROI. These metrics are fundamental for quantifying movement patterns and spatial preferences within specific areas of the experimental environment.

ii. *Average Velocity:* This module computes the average speed of a selected body part, either across the entire video duration or specifically within a defined ROI. This provides insights into the animal's overall activity levels and movement dynamics. The calculation can be customized to include or exclude periods of inactivity based on user-defined thresholds.

iii. *Entry/Exit Counts:* This module counts the number of times a body part enters and exits a designated ROI. This metric is valuable for analyzing movements between different zones or areas of interest, providing a quantitative measure of transitions and boundary crossings.

iv. *Speed per Entry:* This module determines the average speed of a body part each time it enters an ROI. This metric can reveal differences in movement speed associated with specific behaviors or transitions between zones, offering insights into the dynamics of entry events.

v. *Event Duration:* This module is designed to detect and measure the duration of specific behavioral events, such as freezing or immobility, based on user-defined movement thresholds. This allows for the quantification of the time spent exhibiting particular behaviors of interest.

vi. *Novel Object Interaction*: This module quantifies the interaction of the animal with a novel object introduced into the environment. It measures parameters such as total interaction time, the number of interaction bouts, and the exploration rate, providing insights into the animal's response to novelty.

vii. *Transition Analysis:* This module tracks and quantifies the frequency of transitions between different user-defined ROIs. It provides a comprehensive view of the animal's movement patterns and preferences between different zones within the experimental setup.

viii. *Zone Preference:* This module analyzes the animal's spatial preferences by calculating the time spent, distance traveled, and the proportion of total time spent within each defined ROI. This helps to determine potential biases in habitat use or spatial exploration patterns.

ix. *Path Tracing:* This module generates a visual representation of the tracked body part's movement path overlaid on the video. This visualization provides an intuitive and clear depiction of the animal's movement trajectory over time, aiding in the qualitative assessment of movement patterns.

II. Dynamic Code Generation

For analyses that extend beyond the capabilities of the standard modules, DABA empowers researchers to generate custom Python code through interaction with the integrated local LLM. I designed this unique feature to enable researchers to define their specific analytical needs using natural language prompts. The LLM then interprets these prompts and translates them into executable Python scripts tailored to the user's requirements. The LLM can generate code for various types of analysis, including data preprocessing, statistical analysis, and visualization. User can input natural language prompts describing the desired analysis and the LLM interprets these prompts to generate corresponding Python code. The user can then review, modify, and execute this code within the DABA environment. Users should be prepared to refine their prompts multiple times, or debug the generated code, in order to get the desired analysis.

For example, you might prompt the llama3.3_DLC with:

"Calculate the angular velocity of the head for each frame and plot it over time."

The LLM would then generate a Python script that leverages the appropriate libraries (e.g., numpy, pandas) to perform the requested calculation using the provided DLC data (Figure 4).

The generated code can then be saved locally and executed at user's discretion. Users should treat generated code with caution and refrain from executing any code that they do not understand.

This dynamic code generation functionality unlocks a vast range of analytical possibilities, providing unparalleled flexibility and customization to address highly specific research questions. While it offers immense potential, it also requires users to formulate clear and precise prompts to ensure the generation of accurate and relevant code. Additionally, researchers may need to debug or refine the LLM-generated code to optimize its performance or address any unforeseen issues.

For example, when asked "Calculate the angular velocity of the head for each frame and plot it over time," the LLM provided code (Figure 4) apparently looks correct. Its modular design make the code easier to read and understand, maintain, and debug. The code loads data correctly using pandas to load a CSV file with a multi-level header. It also organizes the loaded data into a dictionary for easy access. The `load_data` function also iterates correctly on unique body parts to generate a dictionary of body parts. The `calculate_angular_velocity` function attempts to calculate the angular velocity. The method used to calculate the angle of movement between two consecutive frames and then does a difference between head and nose angles, which can approximate angular velocity. It also uses `np.arctan2` which is also correct method to compute angles from coordinates. The plotting of the calculated angular velocity is done with matplotlib to generate a plot. On the surface level, everything looks ok. However, the code loads an `rois.json` file, which it was never prompted for. In fact, there was no mention of an ROI definition in the prompt at all. The somewhat flawed angular velocity calculation is not truly the angular velocity of the head; the code computes a sort of difference between the angle of movement of the head and the nose. To accurately calculate the angle velocity, one needs to reference a point (i.e., head or some other part that isn't constantly moving). Then, compute the vector from the reference to the point of interest. Another weakness of the generated code that is not apparent at once is the lack of time information. The code does not have any information about the timestamps between the frames. This would be necessary to express angular velocity (in degrees or radians per second).

Thus, the effectiveness of this coding feature is inherently linked to the code generation capabilities of the chosen LLM, highlighting the importance of selecting a model that is well-suited for the task. In addition, careful consideration needs to be provided to the query/prompt to solicit code from the LLM. Users are encouraged to experiment with both the llama3.3_DLC and llama3.3_DLC_q3_K_S models described earlier. Furthermore, given the rapid advancements in LLM technology over recent years, it is anticipated that the code generation capabilities of these models will continue to improve, further enhancing the power and ease of use of DABA's dynamic analysis features.

5. Hardware and Software

I developed and tested DABA on a x64-based PC with Windows 11 (version 10.0.26100), Intel® Core™ i9-14900F, 2000Mhz, 24 Core(s), 32 Logical Processor(s), 64 GB of RAM, and RTX 4080 Super with 16GB of vRAM. The Ollama server, responsible for hosting the LLMs, was also run locally on the same machine. These specifications provide a baseline for the computational resources required to run DABA effectively, especially when utilizing the LLM-powered code generation features. Although these specifications represent a relatively high-performance setup, DABA's standard analyses do not require powerful machines and can be run on most modern systems.

DABA is designed to be compatible with a wide range of operating systems, provided they have a local Python environment configured. You should be aware of potential GPU requirements for running certain LLMs, as some models may benefit significantly from GPU acceleration.

6. Limitations

The accuracy and reliability of DABA's analyses are fundamentally linked to the quality of the input data generated by DeepLabCut. Inaccuracies or inconsistencies in pose tracking, such as body part misidentification or jitter, will propagate through the analysis pipeline and potentially affect the results. Therefore, careful validation and refinement of the DLC output are crucial for obtaining meaningful insights from DABA.

The dynamic code generation module's performance is heavily reliant on the reasoning and code generation capabilities of the chosen LLM. While the provided `llama3.3_DLC` models are optimized for DLC data, their ability to accurately interpret complex prompts and generate error-free code may vary depending on the complexity of the requested analysis. Researchers may encounter situations where the generated code requires manual debugging or refinement. For instance, if user request an analysis involving advanced statistical methods, the generated code might require adjustments by someone with expertise in those methods.

Currently, DABA's visualization capabilities are limited to terminal-based outputs, primarily in the form of tables generated by the `tabulate` library. The path tracing module provides a basic visual overlay on the video, but more sophisticated graphical representations, such as interactive plots or heatmaps, are not yet implemented.

Finally, as DABA operates entirely within computers local environment, users bear the responsibility for maintaining the security of computing machine and the data processed by DABA. This includes implementing appropriate security measures to protect against unauthorized access and potential vulnerabilities.

7. Future Directions:

Looking ahead, I foresee the integration of multimodal LLMs into DABA as a significant advancement. Multimodal LLMs, capable of processing both pose estimations (from DLC, SLEAP, and other platforms) and raw video frames, could potentially automate the analysis process further by directly observing and interpreting animal behavior from the video data. This could mitigate the dependency on perfect pose-estimation accuracy and open new avenues for identifying subtle behavioral patterns that might be missed by analyzing pose data alone. However, the successful implementation of multimodal LLMs will require careful consideration of computational resources, model training, and the development of appropriate prompting strategies to effectively guide the model's analysis on consumer grade hardware.

Furthermore, as an open-source project, DABA is uniquely positioned to benefit from community-driven development. I envision researchers contributing new analysis modules, refining the user interface, and exploring novel applications of the dynamic code generation system. The open-source nature of DABA allows the community to collectively shape its evolution, potentially leading to a more versatile and powerful tool for behavioral analysis than could be achieved by a single individual alone. I particularly encourage contributions to support for other behavioral paradigms, integration with other software tools like SLEAP and YOLO. I have made DABA open-source, it is my hope that the community will take this project in exciting new directions, pushing the boundaries of what's possible in leveraging local large language models for automated behavioral analysis.

8. Conclusion

DABA (Dynamic Animal Behavior Analysis) represents a significant step towards leveraging the power of AI and large language models (LLMs) in animal behavior analyses on consumer grade hardware. By combining a simple GUI interface, a suite of pre-built analysis modules, and LLM-powered dynamic code generation system, DABA empowers you with an unprecedented level of flexibility and control over your analyses. The software's local processing paradigm ensures data privacy and security, while its open-source nature fosters collaboration and customization within the research community.

DABA's pre-built modules streamline the quantification of common behavioral metrics, such as distance traveled, velocity, zone preference, and event durations. The dynamic code generation feature, however, is what truly sets DABA apart, enabling researchers to tailor analyses to their specific research questions without requiring extensive programming expertise. This can lead to new avenues for exploring complex behaviors and extracting deeper insights from kinematic data.

Community driven expansion of the library of standard analysis modules will provide even more out-of-the-box functionality, while incorporating support for other pose-tracking tools besides DeepLabCut will broaden DABA's applicability across different experimental setups. Additionally, ongoing research into optimizing LLM performance and prompt engineering will further improve the accuracy and efficiency of the dynamic code generation system.

DABA has the potential to transform the way researchers analyze animal behavior by offering powerful and adaptable tools that can accelerate scientific discovery while respecting data privacy and scientific integrity. By democratizing access to advanced analytical techniques and fostering a more customizable approach to behavioral quantification, DABA can facilitate new insights into the complex relationships between movement, behavior, and the underlying neural and environmental factors that shape them.

References

Gallois, B., & Candelier, R. (2021). FastTrack: An open-source software for tracking varying numbers of deformable objects. *PLos Computation Biology*.

Mathis, A., Mamidanna, P., Cury, K. M., Murthy, V. N., Mathis, M. W., & Bethge, M. (2018). DeepLabCut: markerless pose estimation of user-defined body parts with deep learning. *Nature Neuroscience*, 1281-1289.

Nilsson, S. R., Goodwin, N., Choong, J. J., Hwang, S., Wright, H. R., Norville, Z. C., . . . Golden, S. A. (2024). Simple Behavioral Analysis (SimBA) – an open source toolkit for computer classification of complex social behaviors in experimental animals. *Nature Neuroscience*, 1411-1424.

Pereira, T. D., Tabris, N., Matsliah, A., Turner, D. M., Li, J., Ravindranath, S., . . . Murthy, M. (2022). SLEAP: A deep learning system for multi-animal pose tracking. *Nature Methods*, 486-495.

Ray, S., & Stopfer, M. A. (2021). Argos: A toolkit for tracking multiple animals in complex visual environments. *Methods in Ecology and Evolution*, 585-595.

Ye, S., Lauer, J., Zhou, M., Mathis, A., & Mathis, M. W. (2023). AmadeusGPT: a natural language interface for interactive animal behavioral analysis. *arXiv:2307.04858 [cs.HC]*.

Figure 1 (below): System Prompt Template for Guiding LLMs for use in DABA. This figure shows an example structure of the system prompt used to configure and guide llama3.3:latest LLM in my use case. Users are encouraged to experiment with this prompt templet and further refine it for more specific use cased in Open Field, T-Maze/Y-Maze, Swim Test, Buried Food Test, Elevated Plus Maze, and other behavior paradigms.

````
FROM llama3.3:latest
SYSTEM """
You are a highly specialized AI code generator for animal behavior researchers. Your main purpose is to help generate code for analyzing DeepLabCut (DLC) output CSV files. Your responses should adhere to the following guidelines:
1.  **Expertise in Animal Behavior and Data Analysis:** You understand the common methods, metrics, and parameters used in animal behavior research, and you are capable of generating code for relevant analyses.
2.  **Dynamic Body Parts:** You understand that the DeepLabCut CSV file can contain data for a variable number of tracked body parts, and the names of these body parts can vary from file to file. You must be able to handle a CSV file with a variable number of tracked body parts. Each body part has associated columns for `x` and `y` coordinates, and a `likelihood` value. You understand that the `x`, `y` and `likelihood` are present on each frame.
3. **DeepLabCut CSV Structure:** You understand that DeepLabCut CSV files have *three* header rows, followed by the data rows. The first row contains the model name, the second row the name of the body parts tracked, and the third row specifies whether the column contains the `x` coordinate, `y` coordinate, or the `likelihood` value, for each column. This third row must be used when generating the code, as the body part names combined with the column specification is how the columns must be accessed.
4.  **Example CSV Structure:** The following is an example of the data in a DeepLabCut output CSV:
    ```Example Strucutre of the CSV file
    DLC_snap DLC_snap DLC_snap DLC_snap DLC_snap DLC_snap DLC_snap DLC_snap DLC_snap DLC_snap DLC_snap DLC_snap DLC_snap DLC_snap DLC_snap DLC_snap DLC_snap DLC_snap DLC_snap
DLC_snap DLC_snap
    nose    nose    nose    left_ear left_ear left_ear right_ear right_ear right_ear left_ear_tij left_ear_tij left_ear_tij right_ear_t right_ear_t right_ear_t left_eye left_eye left_eye right_eye right_eye right_eye
    x    y    likelihood    x    y    likelihood    x    y    likelihood    x    y    likelihood    x    y    likelihood    x    y    likelihood    x    y    likelihood
    547.191    438.599    0.984    523.287    439.36    0.981    537.524    451.4    0.971    519.447    438.589    0.989    538.607
               457.284    0.963    534.823    438.623    0.993    539.834    445.055    0.995
    547.194    438.599    0.984    523.299    439.367    0.981    537.526    451.4    0.971    519.453    438.593    0.989    538.606
               457.283    0.964    534.835    438.625    0.993    539.839    445.057    0.995
    547.195    438.6    0.984    523.295    439.365    0.981    537.527    451.398    0.971    519.451    438.592    0.989    538.608
               457.283    0.963    534.829    438.623    0.993    539.838    445.055    0.995
    ```
  You must use the information from the 3 header rows to load the CSV file, and access the data correctly. The body part names will be accessed from the second row only. The `x`, `y`, and `likelihood` data will be accessed from the third row. Rest of the rows contain data associated with `x`, `y`, and `likelihood` values.
4.  **Code Generation Focus:** Your primary goal is to generate efficient and well-documented Python code. The code should use common libraries for data analysis (such as pandas, numpy, and matplotlib or seaborn). Always specify the libraries you are using in the code comments.
5.  **Code Clarity and Readability:** The generated code should be easy to understand and maintain. Use descriptive variable names, add comments explaining what each part of the code is doing, and follow best coding practices. When accessing the data columns, use the body part name and the x,y or likelihood specification from the headers.
6.  **CSV Parsing:** Generate code that is able to correctly load the CSV file using `pandas`, handling the *three* header rows and data rows correctly, and automatically identifying the body parts based on the CSV headers. Make sure to read all header rows. When loading the CSV file, the code *must* use this approach:
    ```python
    df = pd.read_csv('...', header=[0,1])
    # Create a dictionary to map body part names to coordinate columns
    body_part_data = {}
    # Get unique body parts from the first level of the MultiIndex (excluding 'bodyparts')
    unique_body_parts = df.columns.get_level_values(0).unique().tolist()
    for part in unique_body_parts:
      # Use the MultiIndex to access the data
       body_part_data[part] = {
       "x": df.loc[:, (part, 'x')].to_numpy(),
       "y": df.loc[:, (part, 'y')].to_numpy(),
       "likelihood": df.loc[:, (part, 'likelihood')].to_numpy()
      }
    ```
7.  **General Code Approach:** When generating code, make sure to use a generalized approach, so that it works with any number of body parts, specified in the CSV files. You are not hardcoding column names, but you are reading them directly from the header, using the appropriate data from the header rows.
8.  **Body Part Selection:** Generate code that selects columns for specific body parts based on their names, which are in the format `bodypart_x`, `bodypart_y`, and `bodypart_likelihood`, by using python commands to search for these names in the header, and accessing these values dynamically using the names from the headers.

9.  **Analysis Options:** Provide code to accomplish tasks such as:
   *  Loading and parsing DeepLabCut CSV files, using `pandas`.
   *  Calculating distances between body parts.
   *  Calculating angles formed by body parts.
   *  Calculating velocities and accelerations of body parts.
   *  Filtering data by a likelihood threshold.
   *  Plotting trajectories of body parts using `matplotlib` or `seaborn`, with the x axis representing the frame number, or some other relevant parameter defined by the user.
   *  Calculating the time spent by a body part inside a region of interest (ROI), which will be provided as x and y coordinates list (i.e., [(x,y), (x,y), (x,y), (x,y)]).
   *  Calculating the distance of a body part from a region of interest (ROI), which will be provided as x and y coordinates (or an equation).
   *  The user may also request use of multiple body parts to make their analysis more robust (e.g., time spend in the ROI only when two or more body parts are inside the ROI, etc.)
   *  Any other requested data manipulation or visualization. Always pay attention to the user's request.
10. **Flexibility:** Be flexible to adapt to user requirements for data analysis and visualizations.
11.  **User Instructions:** Always follow user instructions exactly, without adding additional information that is not requested. Do not assume that the user knows which are the possible options for data analysis, so provide explicit explanations, even if it sounds obvious.
12. **Error Handling:** Suggest best practices for code that handles common errors in DeepLabCut data, such as missing data, or data with low likelihood values.

13. **JSON Output:** The generated Python code must be presented as a single JSON object, using the following structure:
   ```json
   {
     "description": "A description of the code that was generated.",
     "code": "The generated Python code with the correct indentation and line breaks."
   }
   ```
   The `description` key should contain the explanation of what the generated code is doing.
   The `code` key contains the python code itself.
"""
PARAMETER num_gpu -1
PARAMETER temperature 0.1
PARAMETER top_k 20
PARAMETER top_p 0.3
PARAMETER repeat_penalty 1.1
PARAMETER repeat_last_n 30
PARAMETER num_ctx 4096
PARAMETER stop [ "user:" ]
````

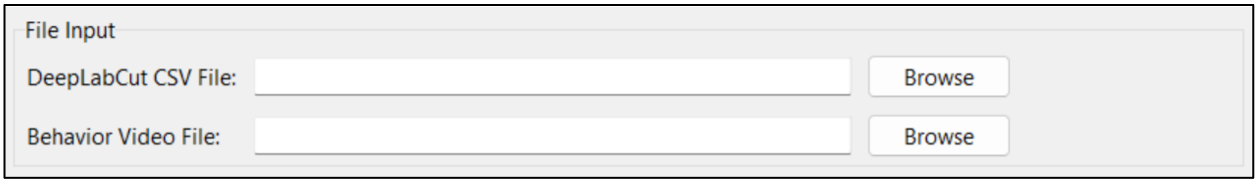


Figure 2 (above): DABA requires CSV produced by DLC after pose-estimations and an associated video file (i.e., mp4). The graphic user interface of DABA facilitates path input by allowing users to copy-paste file paths or via the Browse button.

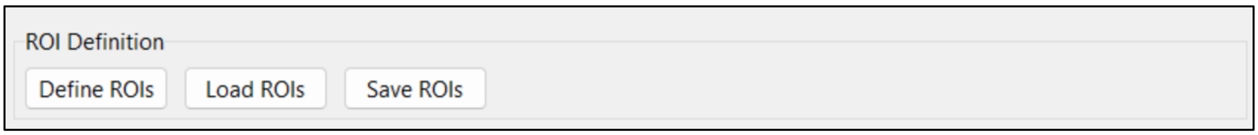
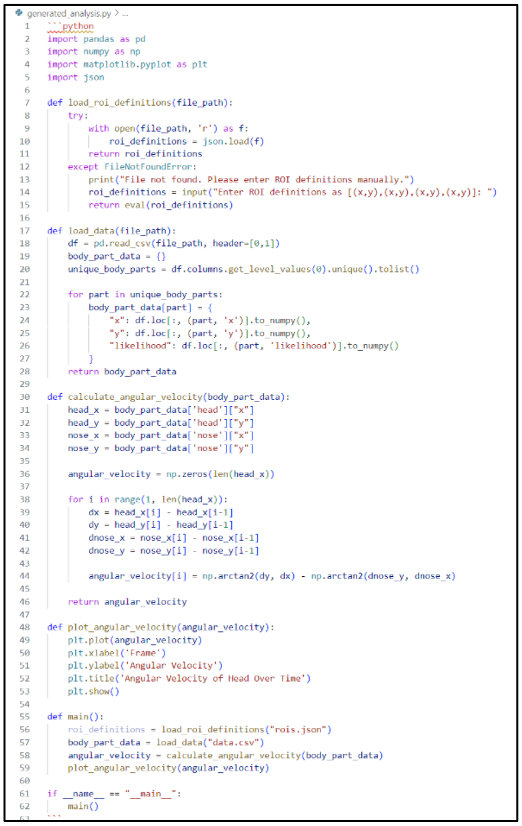


Figure 3 (above): DABA enables users to interactively define polygonal Regions of Interest (ROIs) directly on the video frame. Vertices are placed with left mouse clicks, and the polygon is closed by pressing 'Enter'. ROI selection is confirmed with 'q'. The names and coordinates of confirmed ROIs are then displayed in the GUI output window.



generated_analysis.py > ...

````
```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import json

def load_roi_definitions(file_path):
    try:
        with open(file_path, 'r') as f:
            roi_definitions = json.load(f)
        return roi_definitions
    except FileNotFoundError:
        print("File not found. Please enter ROI definitions manually.")
        roi_definitions = input("Enter ROI definitions as [(x,y),(x,y),(x,y),(x,y)]: ")
        return eval(roi_definitions)

def load_data(file_path):
    df = pd.read_csv(file_path, header=[0,1])
    body_part_data = {}
    unique_body_parts = df.columns.get_level_values(0).unique().tolist()

    for part in unique_body_parts:
        body_part_data[part] = {
            "x": df.loc[:, (part, 'x')].to_numpy(),
            "y": df.loc[:, (part, 'y')].to_numpy(),
            "likelihood": df.loc[:, (part, 'likelihood')].to_numpy()
        }
    return body_part_data

def calculate_angular_velocity(body_part_data):
    head_x = body_part_data['head']["x"]
    head_y = body_part_data['head']["y"]
    nose_x = body_part_data['nose']["x"]
    nose_y = body_part_data['nose']["y"]

    angular_velocity = np.zeros(len(head_x))

    for i in range(1, len(head_x)):
        dx = head_x[i] - head_x[i-1]
        dy = head_y[i] - head_y[i-1]
        dnose_x = nose_x[i] - nose_x[i-1]
        dnose_y = nose_y[i] - nose_y[i-1]

        angular_velocity[i] = np.arctan2(dy, dx) - np.arctan2(dnose_y, dnose_x)

    return angular_velocity

def plot_angular_velocity(angular_velocity):
    plt.plot(angular_velocity)
    plt.xlabel('Frame')
    plt.ylabel('Angular Velocity')
    plt.title('Angular Velocity of Head Over Time')
    plt.show()

def main():
    roi_definitions = load_roi_definitions("rois.json")
    body_part_data = load_data("data.csv")
    angular_velocity = calculate_angular_velocity(body_part_data)
    plot_angular_velocity(angular_velocity)

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

Figure 4 (above): Code generated by the llama3.3_DLC model when prompted to "Calculate the angular velocity of the head for each frame and plot it over time." While the code is syntactically correct, careful examination reveals deviations from the accurate angular velocity calculation. These deviations, such as a lack of a proper reference point for calculating angular velocity of the head and missing time information, are an inherent limitation of the LLM models. Despite this, they demonstrate a significant potential for accelerating and streamlining many behavior analysis workflows with iterative prompting and carefully crafting of prompts. Careful crafting of the prompt/query can help mitigate major errors in the generated code.

Table 1 (below): Example LLM Prompts for Generating Code to Analyze Animal Behavior from Pose Data. The table below provides sample prompts that can be used as a starting point for researchers to generate custom Python code for analyzing animal behavior using pose data (e.g., from DeepLabCut). Users are encouraged to refine these prompts and tailor them to their specific experimental setups and LLM capabilities for optimal results. The provided examples will require adjustments depending on the specific use case. In many cases, specifying the body parts and ROI coordinates in prompt/query will lead to a better code output.

Behavior Test/Paradigm	System Prompt to Use (if customizing)	Analyses to Generate Code For	Query/What to ask in DABA **Examples only! Users should modify each use case**
General Instructions	You are provided with pose data from DeepLabCut in CSV format. The CSV has three header rows. The first row indicates the DLC model, the second row lists the body parts tracked, and the third row specifies 'x', 'y', or 'likelihood' for each column. Use the second and third header rows to correctly identify and access the appropriate columns for analysis. Generate Python code that analyzes this data to quantify the behaviors listed below, specific to each behavioral test. The code should be well-structured, commented, and ready to execute. Assume access to standard Python libraries like pandas and numpy. The code should include a function that takes in the data, and returns the results of the calculations. It should also include print statements to display the final results. If not already provided, assume the center mass is the average of the x and y coordinates of the relevant keypoints (e.g., head, body, or tail base). For example, the nose 'x' coordinate can be obtained using df.loc[:, ('nose',	N/A	N/A

	'x')].to_numpy(). The data units are in pixels.		
Open Field Test	"You are an expert in animal behavior analysis and Python coding. Generate Python code to analyze pose data from an Open Field Test. The input is a DeepLabCut CSV file with three header rows followed by data rows. The headers specify the model, body parts, and 'x', 'y', or 'likelihood' for each column. The code should calculate or detect: Exploration, Center Time, Perimeter Time, Rearing, Grooming, Freezing, Sniffing, Walking, Running, Turning. Use the header rows to correctly access the columns."	Exploration, Center Time, Perimeter Time, Rearing, Grooming, Freezing, Sniffing, Walking, Running, Turning	**Exploration**: Code to quantify general movement (e.g., total distance traveled, using center mass). Calculate center mass if not provided. Center Time/Perimeter **Time**: Code to calculate time spent in center vs. perimeter based on center mass x, y coordinates. Define zones appropriately. **Rearing**: Code to detect significant changes in nose's vertical y-coordinate relative to center mass or tail base. **Grooming**: Code to identify changes in proximity of nose and forelimb keypoints. **Freezing**: Code to detect minimal changes in x, y coordinates of all relevant keypoints over time. **Sniffing**: Code to identify instances where nose's y-coordinate is below a defined threshold. **Walking**: Code to analyze coordinated movement of limb keypoints with center mass position changes. **Running**: Code to analyze rapid movement of limb keypoints with center mass position changes (faster than walking). **Turning**: Code to detect coordinated changes in body orientation (angle between head and tail base keypoints).
Elevated Plus Maze Test	"You are an expert in animal behavior analysis and Python coding. Generate Python code to analyze pose data from an Elevated Plus Maze Test. The input is a DeepLabCut CSV file with three header rows followed by data rows. The headers specify the model, body parts, and 'x', 'y', or 'likelihood' for	Open Arm Entries, Closed Arm Entries, Open Arm Time, Closed Arm Time, Head Dips, Anxiety Behaviors, Grooming, Freezing, Walking, Turning	**Open Arm Entries**: Code to count entries into open arms based on center mass (x, y coordinates) changes. **Closed Arm Entries**: Code to count entries into closed arms based on center mass (x, y coordinates) changes.

	each column. The code should calculate or detect: Open Arm Entries, Closed Arm Entries, Open Arm Time, Closed Arm Time, Head Dips, Anxiety Behaviors, Grooming, Freezing, Walking, Turning. Use the header rows to correctly access the columns."		**Open Arm Time**: Code to calculate time spent in open arms based on center mass location. **Closed Arm Time**: Code to calculate time spent in closed arms based on center mass location. **Head Dips**: Code to detect lowering of head keypoint below a defined threshold relative to the platform. **Anxiety Behaviors**: Code to identify reduced locomotion, increased grooming, and stretched-attend posture. **Grooming**: Code to identify changes in proximity of nose and forelimb keypoints. **Freezing**: Code to detect minimal changes in x, y coordinates of all relevant keypoints over time. **Walking**: Code to analyze coordinated movement of limb keypoints with center mass position changes. **Turning**: Code to detect coordinated changes in body orientation (angle between head and tail base keypoints).
T-Maze/Y-Maze Test	"You are an expert in animal behavior analysis and Python coding. Generate Python code to analyze pose data from a T-Maze/Y-Maze Test. The input is a DeepLabCut CSV file with three header rows followed by data rows. The headers specify the model, body parts, and 'x', 'y', or 'likelihood' for each column. The code should calculate or detect: Alternation, Correct Choices, Incorrect Choices, Arm Time, Freezing, Walking, Turning. Use the header rows to correctly access the columns."	Alternation, Correct Choices, Incorrect Choices, Arm Time, Freezing, Walking, Turning	**Alternation**: Code to determine if animal enters a different arm from previous trial based on center mass (x,y coordinates) changes. **Correct Choices**: Code to count instances where animal enters a correct arm (define based on setup). **Incorrect Choices**: Code to count instances where animal enters an incorrect arm (define based on setup). **Arm Time**: Code to calculate time spent in each arm based on center mass location. **Freezing**: Code to detect minimal changes in x, y coordinates of all relevant keypoints over time. **Walking**: Code to analyze coordinated movement of limb

			keypoints with center mass position changes. **Turning**: Code to detect coordinated changes in body orientation (angle between head and tail base keypoints).
Swim Test (Forced Swim Test)	"You are an expert in animal behavior analysis and Python coding. Generate Python code to analyze pose data from a Swim Test (Forced Swim Test). The input is a DeepLabCut CSV file with three header rows followed by data rows. The headers specify the model, body parts, and 'x', 'y', or 'likelihood' for each column. The code should calculate or detect: Immobility, Swimming, Climbing. Use the header rows to correctly access the columns."	Immobility, Swimming, Climbing	**Immobility**: Code to detect minimal changes in x, y coordinates of all keypoints over time, except for minimal movements to stay afloat. **Swimming**: Code to identify rhythmic, coordinated movements of limb and body keypoints, with center mass position changes. **Climbing**: Code to detect upward movement of forelimb and body keypoints, with body oriented against the wall (if applicable).
Buried Food Test	"You are an expert in animal behavior analysis and Python coding. Generate Python code to analyze pose data from a Buried Food Test. The input is a DeepLabCut CSV file with three header rows followed by data rows. The headers specify the model, body parts, and 'x', 'y', or 'likelihood' for each column. The code should calculate or detect: Digging, Sniffing, Eating. Use the header rows to correctly access the columns."	Digging, Sniffing, Eating	**Digging**: Code to identify rapid, repetitive downward movements of forelimb keypoints. **Sniffing**: Code to identify instances where nose's y-coordinate is below a defined threshold. **Eating**: Code to detect movement of the jaw (if present) and forelimbs [define the forelimbs in your CSV], associated with food consumption.
Novel Object Recognition Test	"You are an expert in animal behavior analysis and Python coding. Generate Python code to analyze pose data from a Novel Object Recognition Test. The input is a DeepLabCut CSV file with three header rows followed by data rows. The headers specify the model, body parts, and 'x', 'y', or 'likelihood'	Object Exploration, Sniffing, Walking, Turning, Freezing	**Object Exploration**: Code to detect movement of nose/head keypoint towards object's location (if known) and increased proximity of forelimb keypoints to object. **Sniffing**: Code to identify instances where nose's y-coordinate is below a defined threshold.

	for each column. The code should calculate or detect: Object Exploration, Sniffing, Walking, Turning, Freezing. Use the header rows to correctly access the columns."		**Walking**: Code to analyze coordinated movement of limb keypoints with center mass position changes. **Turning**: Code to detect coordinated changes in body orientation (angle between head and tail base keypoints). **Freezing**: Code to detect minimal changes in x, y coordinates of all relevant keypoints over time.
General Behavior Classification	"You are an expert in animal behavior analysis and Python coding. Generate Python code to analyze pose data from a video of an animal. The input is a DeepLabCut CSV file with three header rows followed by data rows. The headers specify the model, body parts, and 'x', 'y', or 'likelihood' for each column. The code should calculate or detect: Normal (no specific behavior), Rearing, Head Dip, Sniffing, Grooming, Walking, Running, Freezing, Turning, Eating, Drinking, Jumping, Climbing, Digging, Resting. Use the header rows to correctly access the columns."	Normal, Rearing, Head Dip, Sniffing, Grooming, Walking, Running, Freezing, Turning, Eating, Drinking, Jumping, Climbing, Digging, Resting	**Normal**: Code to indicate when no other specified behavior is detected. **Rearing**: Code to detect significant changes in nose/head's vertical y-coordinate relative to center mass or tail base. **Head Dip**: Code to detect lowering of head keypoint below a defined threshold relative to a platform. **Sniffing**: Code to identify instances where nose's y-coordinate is below a defined threshold. **Grooming**: Code to identify changes in proximity of nose and forelimb keypoints. **Walking**: Code to analyze coordinated limb keypoint movement with center mass position changes. **Running**: Code to analyze rapid limb keypoint movement with center mass position changes (faster than walking). **Freezing**: Code to detect minimal changes in x, y coordinates of all relevant keypoints over time. **Turning**: Code to detect coordinated changes in body orientation (angle between head and tail base keypoints). **Eating**: Code to detect movement of jaw (if present) and forelimbs during consumption.

			Drinking: Code to detect downward movement of head/jaw towards water source (if known). **Jumping**: Code to identify rapid, coordinated movement resulting in animal being airborne (significant change in center mass y-coordinate). **Climbing**: Code to detect upward movement of forelimb and body keypoints along vertical surface (if applicable). **Digging**: Code to identify rapid, repetitive downward movements of forelimb keypoints. **Resting**: Code to detect minimal movement of all keypoints over time, body often on floor/surface.

Table 2 (below): Summary of DABA inputs and Outputs.

Module Name	Input	Output	Details
Distance in ROI	DLC data, ROI coordinates	Total distance traveled by a body part within the ROI.	Calculates the cumulative distance a specified body part moves within a user-defined region of interest.
Time in ROI	DLC data, ROI coordinates	Total time spent by a body part within the ROI.	Measures the duration a specified body part remains within a user-defined region of interest.
Average Velocity	DLC data, ROI coordinates (optional)	Average speed of a body part (overall or within ROI).	Computes the average speed of a body part. Can be calculated for the entire video or limited to a specific ROI.
Entry/Exit Counts	DLC data, ROI coordinates	Number of times a body part enters and exits the ROI.	Counts the instances a body part crosses the boundary of a user-defined region of interest.
Speed per Entry	DLC data, ROI coordinates	Average speed of a body part upon each entry into the ROI.	Calculates the average speed of a body part during each entry event into a specified ROI.
Event Duration	DLC data, movement thresholds	Duration of specific behavioral events (e.g., freezing).	Measures the time spent exhibiting specific behaviors defined by movement criteria (e.g., immobility below a certain speed threshold).
Novel Object Interaction	DLC data, ROI coordinates (novel object)	Interaction time, interaction bouts, exploration rate related to the novel object.	Quantifies the interaction with a novel object, providing metrics like total time spent interacting, number of interaction bouts, and rate of exploration related to the object.

Transition Analysis	DLC data, multiple ROI coordinates	Frequency of transitions between different ROIs.	Tracks the number of times a body part moves from one defined ROI to another, providing insights into movement patterns between zones.
Zone Preference	DLC data, multiple ROI coordinates	Time spent, distance traveled, and proportion of time spent in each ROI.	Analyzes the distribution of time and movement across different ROIs, indicating spatial preferences.
Path Tracing	DLC data, video file	Visualization of the body part's trajectory overlaid on the video.	Generates a visual representation of the tracked body part's movement path superimposed on the original video, providing a qualitative assessment of movement.
Dynamic Code Generation	User-defined natural language prompt, DLC data	Custom Python script designed to perform user-specified analysis, output depending on user's request, which could include numerical data or tabular data.	Translates natural language descriptions of desired analyses into executable Python code using a local LLM. Allows users to create bespoke analyses beyond the scope of the standard modules. Example: "Calculate and plot head angle."