**User Manual**

**System requirements:**

**Hardware:**

Trackball setup with two optical mouse sensors for reading data, a functional network connection between the computer with the visual stimuli software and the computer with FlyTracker. A network connection is only mandatory when using network trigger.

**Software:**

* Ubuntu Linux (definitely works on: 10.04 LTS and 12.04 LTS, none other tested)
* Matlab (definitely works on: R2012a none other tested)

**Installing and setting up Git:**

**To install git on the computer run the following commands in the ubuntu terminal:**

**sudo apt-get update**

**sudo apt-get install git**

**(if the last line doesnt work try changing git to git-core)**

**Now you need to setup git so that you can access you account:**

**git config --global user.name motionvision**

**git config --global user.email karin.nordström@neuro.uu.se**

**Git should now be set up correctly, to retrieve a project for the first time use:**

**git clone “repository clone address”**

**The clone address can be found on the project page at github.com under “clone url”**

**After making changes, to upload them:**

**git commit -a**

**Then follow instructions and finalize the upload by:**

**git push**

**If you created new files that already werent in FlyTracker add them to the repository by the following line (before commiting)**

**git add “filename”**

**(The filename is the relative path from the FlyTracker folder but can also be found when running “git commit -a” if the file has not yet been added)**

**To reset to a previous version of FlyTracker (not recommended for novice users):**

**Run command:  
  
git log**

**There a list of all commits will be shown. Each commit point will look something like this:**

**commit 5f53127372cb1cc0b1adac31ef12a16ab8ddc6e0**

**Author: jakelamotta <kristian.johansson86@gmail.com>**

**Date: Thu Aug 21 19:09:29 2014 +0200**

**Find the point you want to reset to and copy the code after commit and run the following command (with the above commit point as example)**

**git reset 5f53127372cb1cc0b1adac31ef12a16ab8ddc6e0**

**git pull**

**Setup:**

Put the FlyTracker-folder anywhere on the desktop. Add the path, with all subfolders in Matlab.   
  
FlyTracker reads from and writes to files frequently. Therefore relative paths are set within the FlyTracker folder and it is very important that files are not moved manually. Only exception to this rule is the data folder which contains the data files. You can also set the save path for data files in the main GUI.

The path to the FlyTracker folder also has to be manually set in utilities.py (located in the python subfolder). It only needs to be added in beginning of the file.

**Calibration**

Calibrating the setup is the most complex task. This is done in six steps:

1. Password – The user needs to enter their user login password as this is needed to get root access, which in turn is needed to read data from the mice. This is a potential error source as an incorrect password will render the program useless.
2. Number of mice – Make sure that there are exactly three mice (the two used as sensors and the desktop mouse) connected to the computer.
3. ID-mouse – In this step the user is prompted to press the button “Identify mouse” and then move the desktop mouse. Here an error message will occur if for example the user entered the incorrect password in step one. The user will be shown either a success message or an error-message.
4. Ball settings – Enter the diameter of the ball used in the setup and the angle between forward and sensor one, denoted as “a” in the pictogram. The arrow in the pictogram is defined as the fly's forward direction. The calibration adjusts for the fact that the ball rotating in the forward direction actually corresponds to the fly walking backwards.
5. Calibrate translation – The first calibration step where the user calibrates translation motion of the fly (thrust and sideslip). If the angle “a” in the pictogram in the previous step is set so that a mod 45 = 0 both sideslip and thrust movement are recommended to be calibrated on its own as each will only activate one of the sensor’ s readings. Using any other angle “a” it is sufficient to calibrate for either thrust or sideslip.

The distance that is used needs to be the same for each run, though it can vary between sideslip and forward. The ball needs to be rotated away from the sensors, if not the positive and negative direction will be flipped.

1. Calibrate yaw – The second calibration step. Here the user needs to enter the rotation distance in degrees. The ball should then be rotated counter-clockwise (this makes clockwise the positive direction in the output). If you rotate the ball the other direction negative and positive directions will be flipped.

To be able to run the program for actual experiments the software needs to be correctly calibrated. Before calibration it is not possible to perform experiments. This is implemented as a safety measure.

After calibration is done, it is possible to edit certain parts of the calibration. This can be done by redoing one specific part without having to go through all steps again. You can, for example, change the angle of the fly in relation to the sensors, or change the size of the ball. You can also redo the calibration for either translation or yaw, respectively.

**Trigger**

The trigger is what defines when FlyTracker is supposed to start recording and when it is supposed to stop. There are three types of triggering methods, either the user can choose no trigger at all, a timer or a network trigger. These three alternatives are described in more detail below.

**No Trigger**

The system can be run without a trigger, either by having the user manually start and stop each recording, or by setting a timer for how long FlyTracker should record.

**Setting up the network trigger client**

The simple trigger client is implemented in python and is provided as the file TriggerClient.py. This file needs to be called from the visual stimulus software to ensure that commands are sent over the network at the correct time.

Currently the Python client supports communication via a named pipe. This is a sufficiently fast solution for inter process communication that also allows many different platforms for the visual stimulus implementation.   
  
The server accepts three types of commands: “start”, “pause” and “quit”. Needless to say, you first need to start the recording by sending “start”. The “pause” command is used to pause recordings, this will also lead to the creation of a new block as the recording is restarted by providing “start” as a command.

To be able to run FlyTracker with a network trigger three things needs to be configured:  
  
1. Host: Set this to the IP-address or the hostname of the FlyTracker computer, the information can be found in the menu of the main window of FlyTracker. Choose Settings>Network Settings.

Use either hostname or IP-address.

2. Port: Use the same port number in the client as in FlyTracker. This can be changed but ensure that the new port is open and valid. The port number normally needs to be larger than 1024 and lower than 65534, but this may vary between platforms.

3. Pipe: The pipe in TriggerClient.py needs to have the same filename and path as the pipe created in the visual stimulus software. To create the pipe the following matlab code can be used:

Creating the pipe and starting TriggerClient.py:

*pipe = '/home/Username/pipe';  
system('mkfifo /home/Username/pipe');*

*system('python /home/..../TriggerClient.py');*

When drawing of visual stimulus starts:

*fid = fopen(pipe,'w');*

*fwrite(fid,'start');*

*fclose(fid);*

After drawing is done:

*fid = fopen(pipe,'w');*

*fwrite(fid,'pause');*

*fclose(fid);*

Actually, using the quit command is not necessary since you can use “stop” in FlyTracker. This stops the recording and closes down the network connection.

**Saving data**

The data is saved into mat-files. Each file is initialized by pressing “Run” in FlyTracker and saved when the user presses “Stop”. Everything that is recorded in between is saved into the same data file. If you are using network trigger, recordings can also be paused and each pause then creates a new data block within the same mat-file.

Data is saved into .mat-files with the filename of the current time. Each data file consists of one 5 x n cell where n is the number of blocks. The elements of the first row in the cell consists of arrays of thrust delta position, second row is for sideslip delta position, third is for yaw delta position and the forth one is a time stamp. Each column represents data for that particular time stamp. The final row of the cell holds the starting time of that block, i.e. not an array but rather a date and a time.

**Parameter merging**

**Correlating the flyfly stimuli parameters with the correct FlyTracker datablock is now done automatically and in real time. After each experiment in flyfly is done a struct object with all the needed parameters are sent over the network to the FlyTracker computer. In mergeClient.m in the matlab/model folder the network communication is taken care of. Here the user sets the correct IP-address (flyfly computer) and port (same as in flyflys sendParam.m).**

**The parameters of each datablock are saved in the 6th row.**

**Should parameter merging not be used it needs to be disabled both in flyfly and FlyTracker otherwise they will not work correctly.**

**Plotting**

There are four plots, the three first ones are thrust, sideslip and yaw and the user can in settings select whether to plot velocity, position over time or change in position. The fourth plot displays a 2D map of the fly's path.

When using the network trigger data are potentially divided into blocks. In this case not all the data will be displayed but only the latest block.

After stopping the recording the user can go back and look at older blocks using the drop-down menu.

Plotting is done in pseudo-real time by redrawing the plot every 100 data points. This can be changed, but only by changing the hardcoded value in the readdata.m file.

**Calculation of the fly's movement**

The algorithm to calculate the mouse inputs to fly walking movement is pretty straightforward. The main algorithm is copied from “FicTrac: A visual method for tracking spherical motion and generating fictive animal paths”. The ball movement measured by the horizontal component of each mouse sensor is transformed into the yaw component of the fly's movement, while the vertical components of the sensors are used to calculate translation.  
  
Some issues have been detected regarding yaw rotation. For example, it is important that the ball is at such height that the sensor is aimed at the center of the ball. Otherwise a significant yaw component will be measured even doing pure thrust and sideslip movement of the fly.

To further reduce this issue an algorithm which always picks the smallest absolute value between the two horizontal readings (unless one is zero in which case the average is used) and the average of the two has been implemented. No scientific testing has been applied to test whether it actually reduces the issue or not, but from what we can see it does.

**Known issues:**

* Sometimes one of the sensors seem to stop reading. I am not sure why this is the case, nor do I know if it is a hardware issue or a software issue. However, it does seem to be solved by disconnecting the mice for example which would suggest it is a software issue.
* **If FlyTracker is closed during a recording or a recording is stopped incorrectly for example by using CTRL-C the python process spawned by FlyTracker will not be closed. If then FlyTracker is restarted and recordings are made the data will be corrupt. To solve this close Matlab or if you are familiar with how to close processes from the terminal window kill python from there. Symptoms of this is clear, since the plot will be nonsensical.**
* As the main application is in Matlab, the users won't be explicitly warned about any exceptions in the python process. Therefore, there is a log-file in the data-folder where all thrown exceptions are saved. If FlyTracker is acting strangely, having a look in this txt file as it could provide answers.

**For advanced users:**

**The Python-Matlab interface**

The main application uses both Matlab and python code. The communication between them is limited as much as possible and is exclusively done via system calls reading from and writing to named pipes.

**Matlab package**

Given Matlab's weak support for object oriented programming (and because I was lazy) there is a high level of coupling in the matlab code but a division into interface files and data processing files has been done.

**Model**

Holds all files dealing with reading and processing data.

**View**

The view holds all files regarding presenting the data and handling user interactions.

**Python package**

**DAQ file**

The file where most of the work is done, it starts the recording, saves the mouse data to a temporary data file which is then parsed in Matlab. The classes of the DAQ-file are:

**MouseHandler**

This class deals with instantiating the mice, inherits from Thread and runs a loop that evaluates the mice coordinates and saves them to file anytime data is read. Data is also time stamped here. MouseHandler thereby runs three threads, its own loop that saves data and the two sensor-mice in separate threads.

**Mouse class tree**

Currently two types of mice, SensorMouse and IdMouse which inherits from AbstractMouse which in turn is of type Thread. The SensorMouse runs an infinite loop reading from its mouse and write the results into the dictionary holding the coordinates. IdMouse is just used to id the mice used as sensors and the mouse used as an actual mouse.

**MouseID**

This file holds the code for identifying the connected mice. It is dependent on three mice being connected to the computer and that one of them is the desktop mouse and the other two are sensors.

**Utilities**

A utility class providing static methods for different kinds of data manipulation, file I/O and exception logging for debugging.