tvaLib Training Manual

v. 0.58

Paul G. St-Aubin

Monday, March 19, 2018

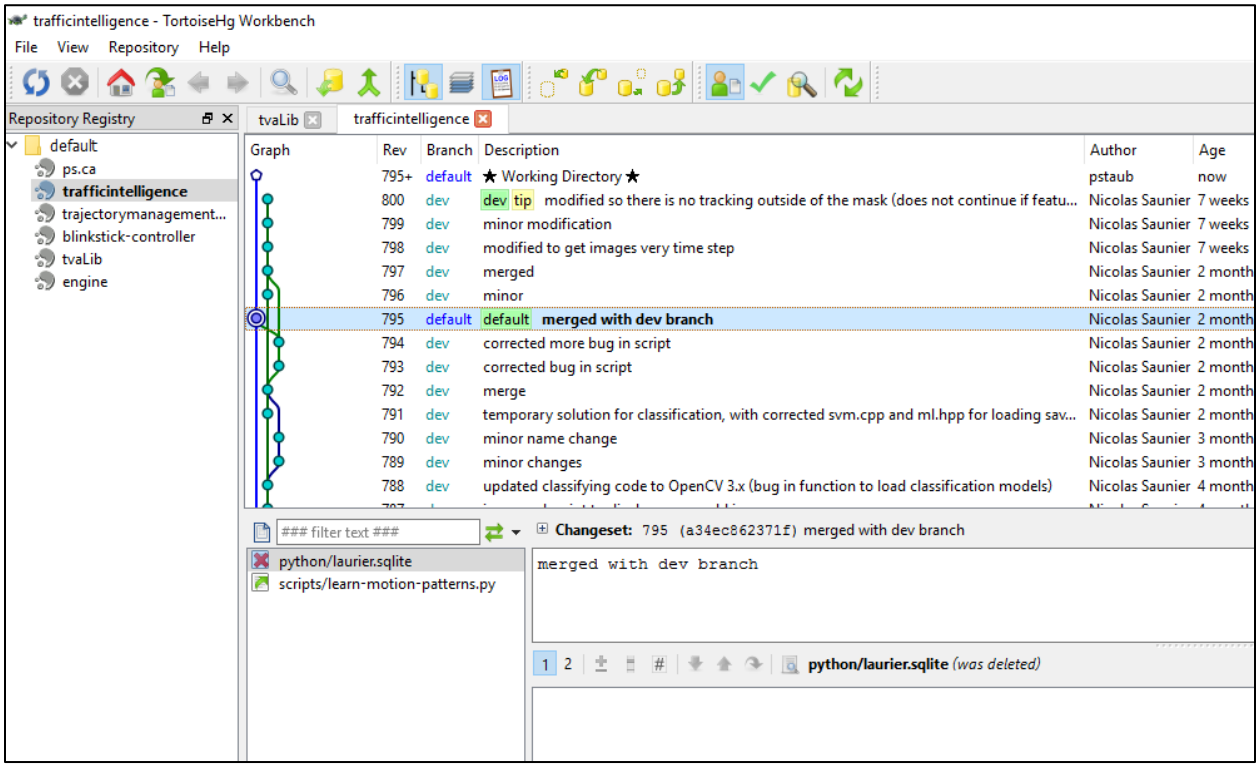
1. Table of Contents

2. Install Traffic Intelligence and tvaLib

2.1. Windows

Mercurial

It is highly recommended to install Traffic Intelligence and tvaLib through CVS utilities, specifically Mercurial CVS. Each packages repository makes use of Mercurial. Therefore, the first thing that you should install is Mercurial. For Windows users who are familiar or predominantly comfortable with GUI environments, the GUI tool, Tortoise HG, is available and recommended (the installer includes Mercurial). It looks as such:



As an alternative to using Mercurial, the source code can be downloaded and extracted manually from its repository. However, it may be tedious to push updates using this method.

Python

Install Python. If your OS is 64-bit, you will want to install 64-bit Python 2.7. Download and run the Windows x86-64 MSI installer from here: https://www.python.org/downloads/release/python-2712/

The editor Spyder is a recommended working environment on Windows, especially if coding will be undertaken. It can optionally be downloaded and installed from here: https://github.com/spyder-ide/spyder/releases

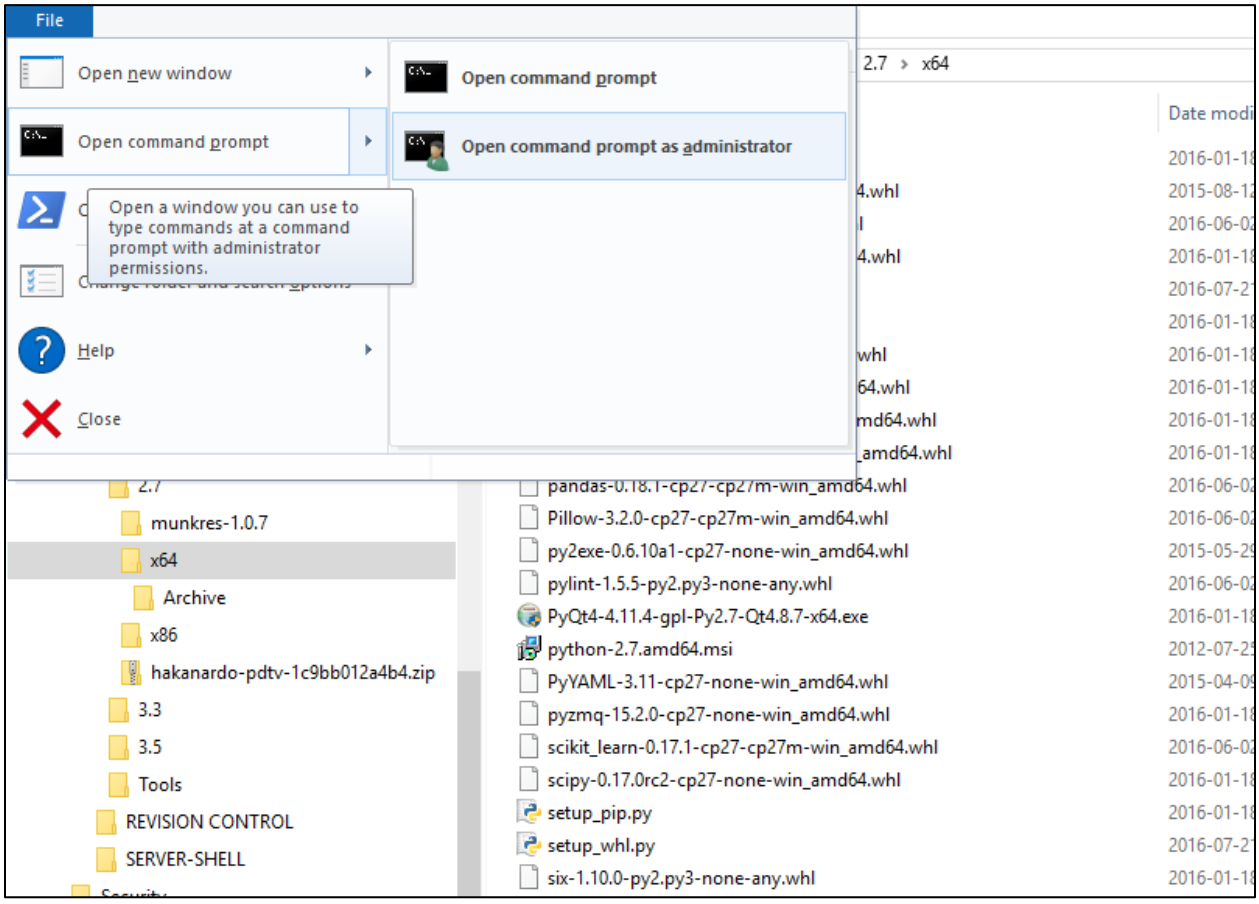
Next, you will need to install the required python modules. The easiest installation method is to use the `pip` command in a Windows command prompt:

```
pip install PACKAGE_NAME
```

If PIP is not already installed, bootstrap it with instructions found on this page:
https://pip.pypa.io/en/latest/installing/#install-pip

The recommended (**required**) python modules to be downloaded and installed include:

basemap	Pillow
colorama	py2exe
cx_Freeze	pylint
ipython	pyYAML
jupyter	pyzmq
matplotlib	**scikit**
numpy	**scipy**
opencv	six
pandas	**SQLAlchemy**



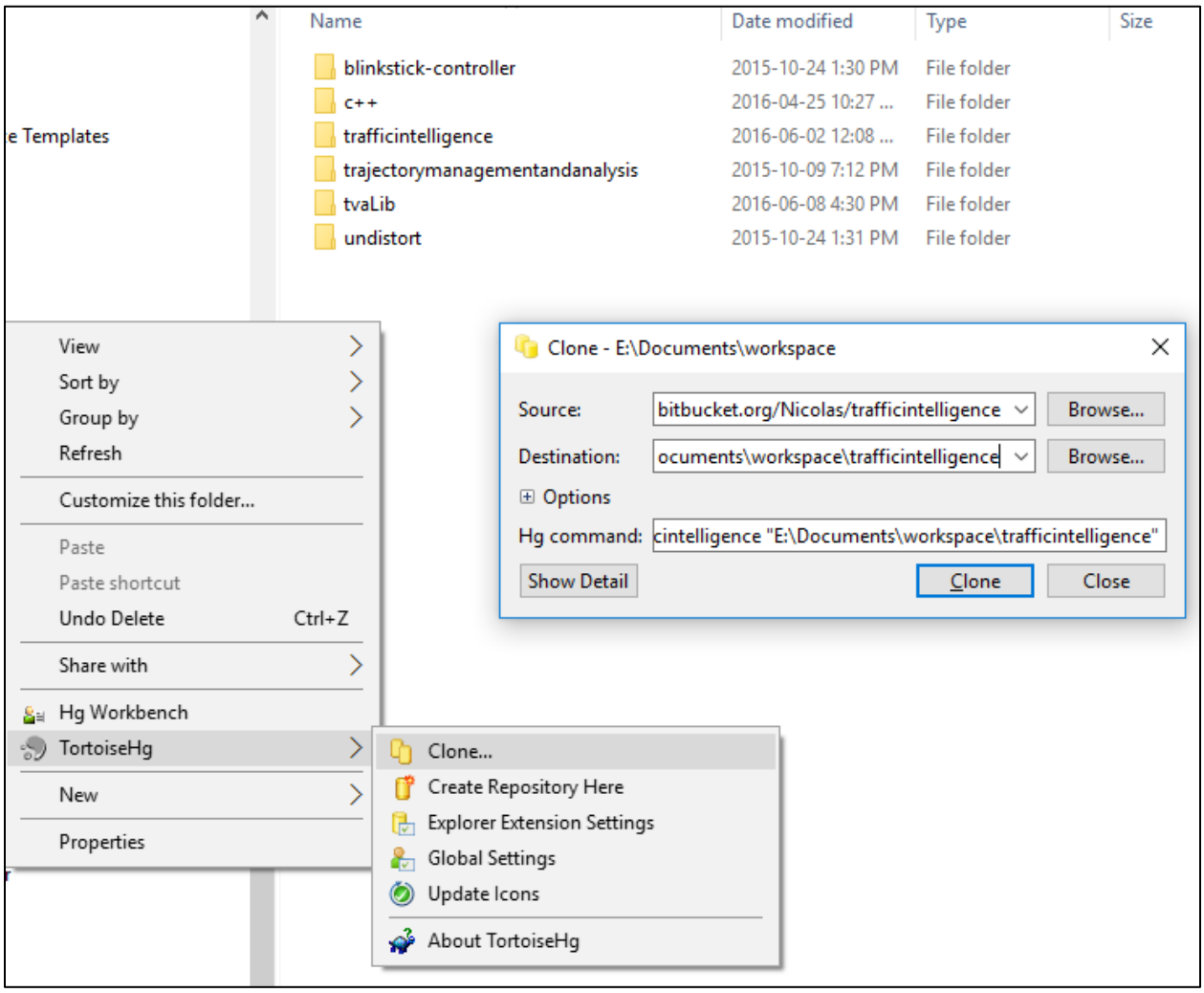
Alternatively, on Windows 64-bit machines, precompiled binaries can also be obtained from:
http://www.lfd.uci.edu/~gohlke/pythonlibs/

To quickly install all downloaded precompiled packages, place the included script `setup_whl.py` in the same download directory as all of these python modules (`.WHL` files) and in a Windows command

prompt, run `setup_whl.py`. You may have to run this script twice, and as an administrator, to properly install all modules.

Traffic Intelligence

Choose a directory in which to install Traffic Intelligence and tvaLib. With TortoiseHG installed, `right-click > TortoiseHG > Clone…` and then copy the following path: "https://bitbucket.org/Nicolas/trafficintelligence" into `Source:`.



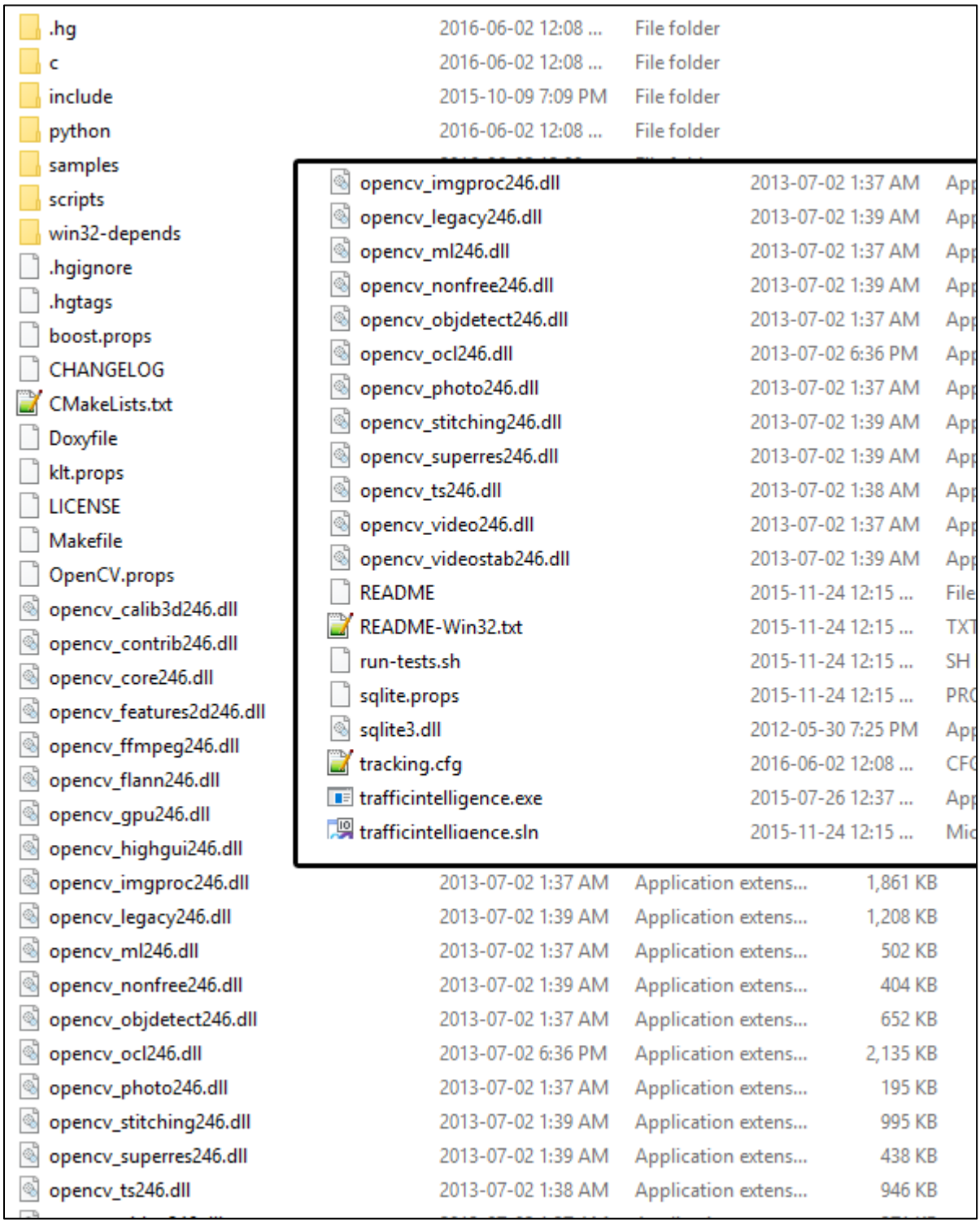
Alternatively, run the following command in a Windows command prompt from within the desired directory:

```
hg clone https://bitbucket.org/Nicolas/trafficintelligence
```

Alternatively, the source code can be downloaded and extracted manually from the repository webpage.

Finally, download the most recent Traffic Intelligence Windows compilation from here: https://bitbucket.org/Nicolas/trafficintelligence/downloads and unzip it in the

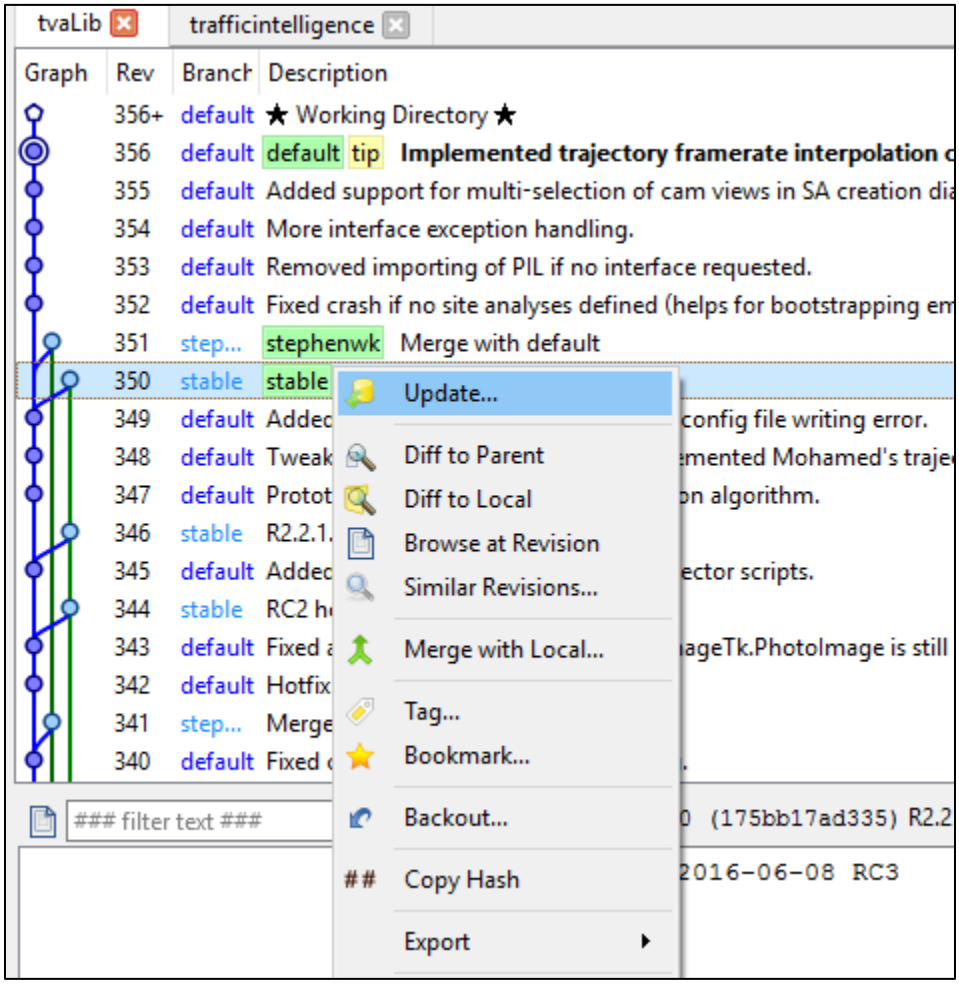
`trafficintelligence` folder. Once this operation is performed, the target folder should look something like:



tvaLib

Repeat the previous Mercurial (or manual) steps using https://bitbucket.org/pstaub/tvalib as source. No source compilation or binary downloads are required.

It is recommended to work in the `stable` branch. To change branches in Mercurial, right-click the top-most branch titled `stable` and select `Update…`. This will change the code in the working directory to use whatever code is in the stable branch:

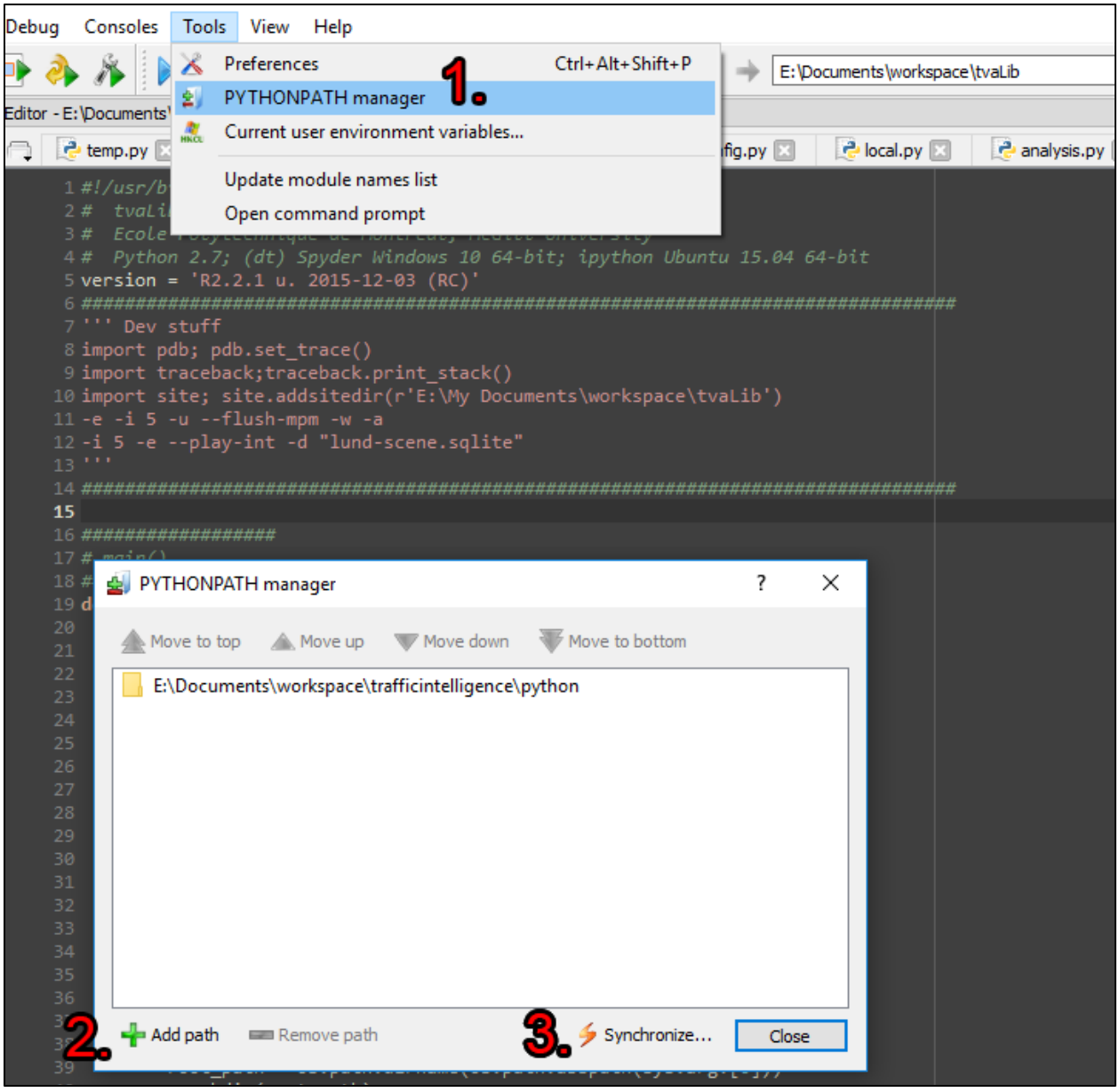


Setup

You must add the `../trafficintelligence/python` path to your list of PYTHONPATHS.

The easiest way to do this is through Spyder. Follow the steps in the following image, then log out of your Windows session and then log back in (alternatively, just reboot the computer).

If Spyder is not installed, follow these instructions instead:
https://stackoverflow.com/questions/3701646/how-to-add-to-the-pythonpath-in-windows-7



Next, find, or designate, an appropriate location to store the entire video database. This is typically done on a bare drive used for the sole purpose of video analysis. The data will be organised in a specific manner as described in section 2. Video Data Storage, and will contain a SQLite file that indexes all of the data. This path usually looks something like: `I:\Video\scene.sqlite`

Run `main.py` once (located in the tvaLib folder). If all python modules are installed, this first launch will generate a new configuration file (`tva.cfg`). It will ask you for the location of `../trafficintelligence/` and it will ask you for the location of the `scene.sqlite` file. Point the program to these files.

If `main.py` crashes at this stage, then there is a problem with you Python installation.

2.2. Linux

Note that the following instructions are heavily geared towards Debian-based Linux distributions, particularly Ubuntu. The specific commands may vary slightly for other Linux distributions.

Mercurial

It is highly recommended to install Traffic Intelligence and tvaLib through CVS. Both of these make use of Mercurial CVS. Mercurial can be installed from the terminal with the command:

```
sudo apt-get install mercurial
```

Python

Python comes preinstalled on most distributions of Linux. It may be required to install some of the following modules (the most important packages are in **bold**). Some of the packages need to be installed via `apt-get` due to build dependencies:

python-matplotlib	**python-opencv**

Others may be installed using `pip`. To bootstrap `pip` installation, in a terminal, use the command:

```
sudo apt-get install python-pip
```

numpy **pandas** **scikit-learn**	**scipy** **SQLAlchemy**

To install packages using `pip`, in a terminal, use the command:

```
sudo pip install <package_name>
```

OpenCV

OpenCV 2.4.x is required for Traffic Intelligence at this time. To install openCV 2.4.x in Linux, you will have to build it from source, even though openCV packages exist for Linux.

You must first install the dependencies. In a terminal, use the following commands:

```
sudo apt-get install build-essential cmake git libgtk2.0-dev pkg-
config libavcodec-dev libavformat-dev libswscale-dev libavresample-dev
```

Then, in a terminal, use the following commands:

```
wget https://github.com/Itseez/opencv/archive/2.4.13.zip

unzip 2.4.13.zip

cd opencv-2.4.13

mkdir build

cd build
```

```
cmake ..

make

sudo make install
```

If you need to uninstall openCV for some reason, from within the `../build/` directory, in a terminal, use the command:

```
 sudo make uninstall
```

For more information, follow the complete instructions to compile openCV on Linux here: https://bitbucket.org/Nicolas/trafficintelligence/wiki/Compile%20the%20C++%20Code

Library for Trajectory Management

Choose a directory in which to install the Trajectory Management library, then run the following command in a terminal from this directory:

```
hg clone
https://bitbucket.org/trajectories/trajectorymanagementandanalysis
```

You must then install the dependencies. In a terminal, use the following commands:

```
sudo apt-get install sqlite3 libsqlite3-dev ccmake
```

Next, to compile the library, from the `..trajectorymanagementandanalysis/trunk/src/TrajectoryManagementAndAnalysis/` directory, in a terminal, run the following commands:

```
ccmake .

make TrajectoryManagementAndAnalysis
```

For more information, follow the instructions to compile the Trajectory Management library on Linux here: https://bitbucket.org/trajectories/trajectorymanagementandanalysis/wiki/Home

Traffic Intelligence

Choose a directory in which to install Traffic Intelligence, then, from that directory, run the following command in a terminal:

```
hg clone https://bitbucket.org/Nicolas/trafficintelligence
```

The `../trafficintelligence/c/MakeFile` file should be edited (in a text editor). Edit line 3 containing the following:

```
TRAJECTORYMANAGEMENT_DIR=$(HOME)/Research/Code/trajectorymanagementand
analysis/trunk/src/TrajectoryManagementAndAnalysis
```

It should point to the equivalent location of your Trajectory Management Library installation, i.e. edit the `$(HOME)/Research/Code` part.

The simplest method of building Traffic Intelligence is to run the following commands from within the directory `../trafficintelligence/c/`:

```
make feature-based-tracking

sudo make install
```

For more information, follow the instructions to compile Traffic Intelligence on Linux here: https://bitbucket.org/Nicolas/trafficintelligence/wiki/Compile%20the%20C++%20Code

tvaLib

Repeat the previous Mercurial steps using https://bitbucket.org/pstaub/tvalib as source. No source compilation is required.

It is recommended to work in the `stable` branch. To change branches in Mercurial, run the following command in a terminal from the repository directory:

```
hg up stable
```

Setup

You must add the `../trafficintelligence/python/` path to your list of PYTHONPATHS. To do so, add the following line to the end of your `~/.bashrc` file using a text editor (it is important to point to the `python/` directory, and not the root `trafficintelligence/` directory):

```
export
PYTHONPATH="${PYTHONPATH}:/path/to/traffic/intelligence/python/"
```

To enact this change immediately, run the following command in a terminal:

```
source ~/.bashrc
```

Next, find, or designate, an appropriate location to store the entire video database. This is typically done on a bare drive used for the sole purpose of video analysis. The data will be organised in a specific manner as described in section 2. Video Data Storage, and will contain a SQLite file that indexes all of the data. This path usually looks something like: `/media/Video/scene.sqlite`

At this step, and every time tvaLib is updated, you may have to make `../tvalib/main.py` executable. To do so, from the `../tvalib/` directory, in a terminal, use the following command:

```
chmod +x main.py
```

Run `main.py` once (located in the tvaLib folder). If all python modules are installed, this first launch will generate a new configuration file (`tva.cfg`). It will ask you for the location of `../trafficintelligence/` and it will ask you for the location of the `scene.sqlite` file. Point the program to these files.

If `main.py` crashes at this stage, then there is a problem with you Python installation.

3. Video Data Storage & Indexing

3.1. Database Location

Whether using Traffic Intelligence or tvaLib, video data **MUST** be stored following a specific format encoded in the `scene.sqlite` file. To bootstrap a new `scene.sqlite`, navigate to the `../tvalib/scripts/` directory and run the following command from a terminal or command prompt:

```
python create-metadata.py -z "ROOT_PATH_TO_VIDEO_DB_DIRECTORY"
```

Where `ROOT_PATH_TO_VIDEO_DB_DIRECTORY` is the root path to the video data repository (where `scene.sqlite` is to be stored), e.g. `I:\Video`

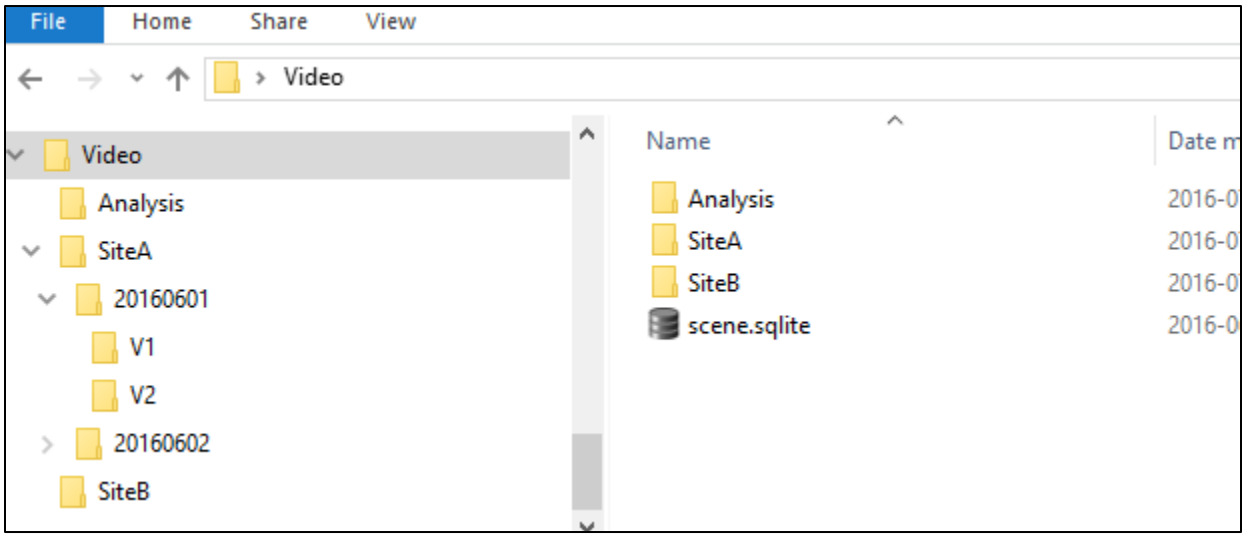
3.2. Data Storage Structure

Organisation

Data for each physical location, a **site**, where filming takes place is to be placed in a separate subfolder of the root path to the video data. Each **camera**, a grouping of video **sequences** taken in succession without moving the field of view of the camera, is also placed in a subfolder within the site subfolder. The camera subfolder structure is arbitrary, and can have many sub-sub folders, as long as all sequences of the camera reside in the same folder. By convention, cameras are split into filming days, and subdivided again into field of views (if multiple exist).

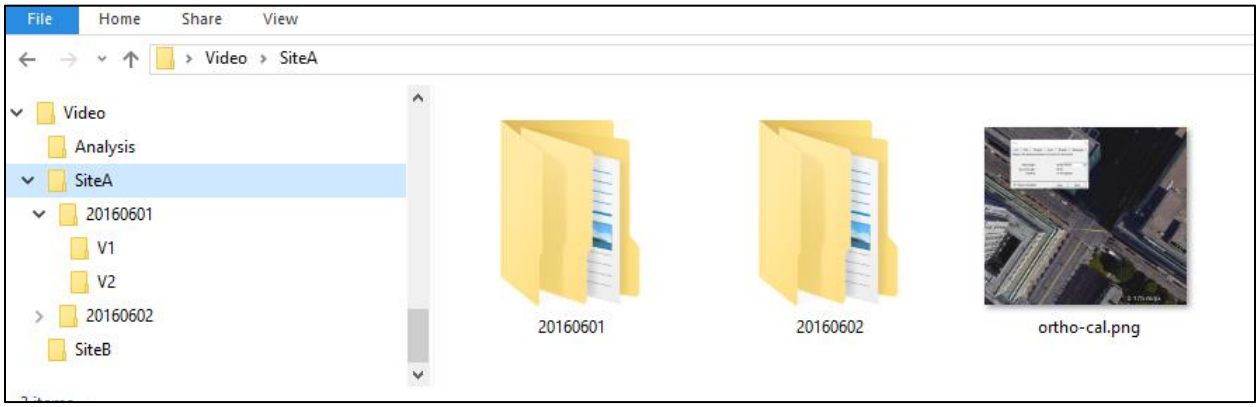
For example, data is collected at two sites: "SiteA" and "SiteB". At SiteA, filming took place on July 1st and then again on July 2nd. On each of these days, two cameras were installed (simultaneously at SiteA). Thus, four cameras are created for SiteA. These are named:

```
20160601/V1
20160601/V2
20160602/V1
20160602/V2
```

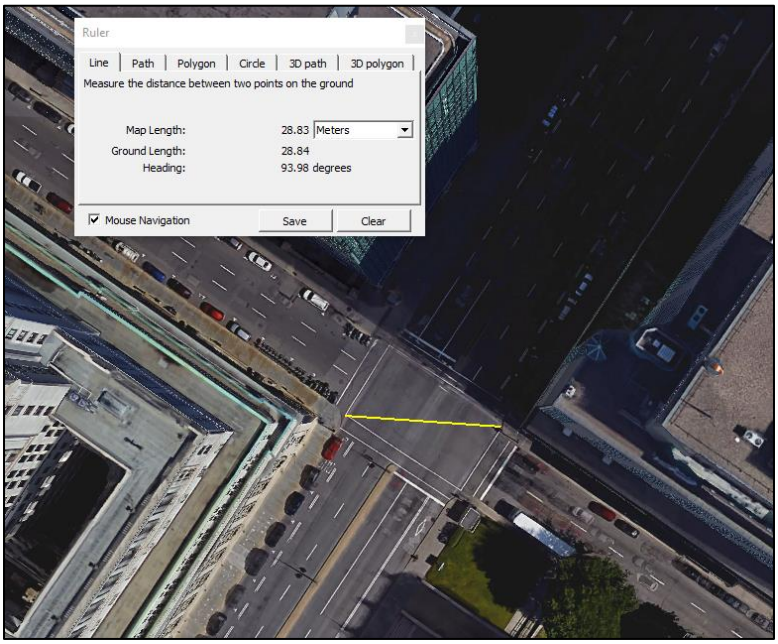


Satellite files

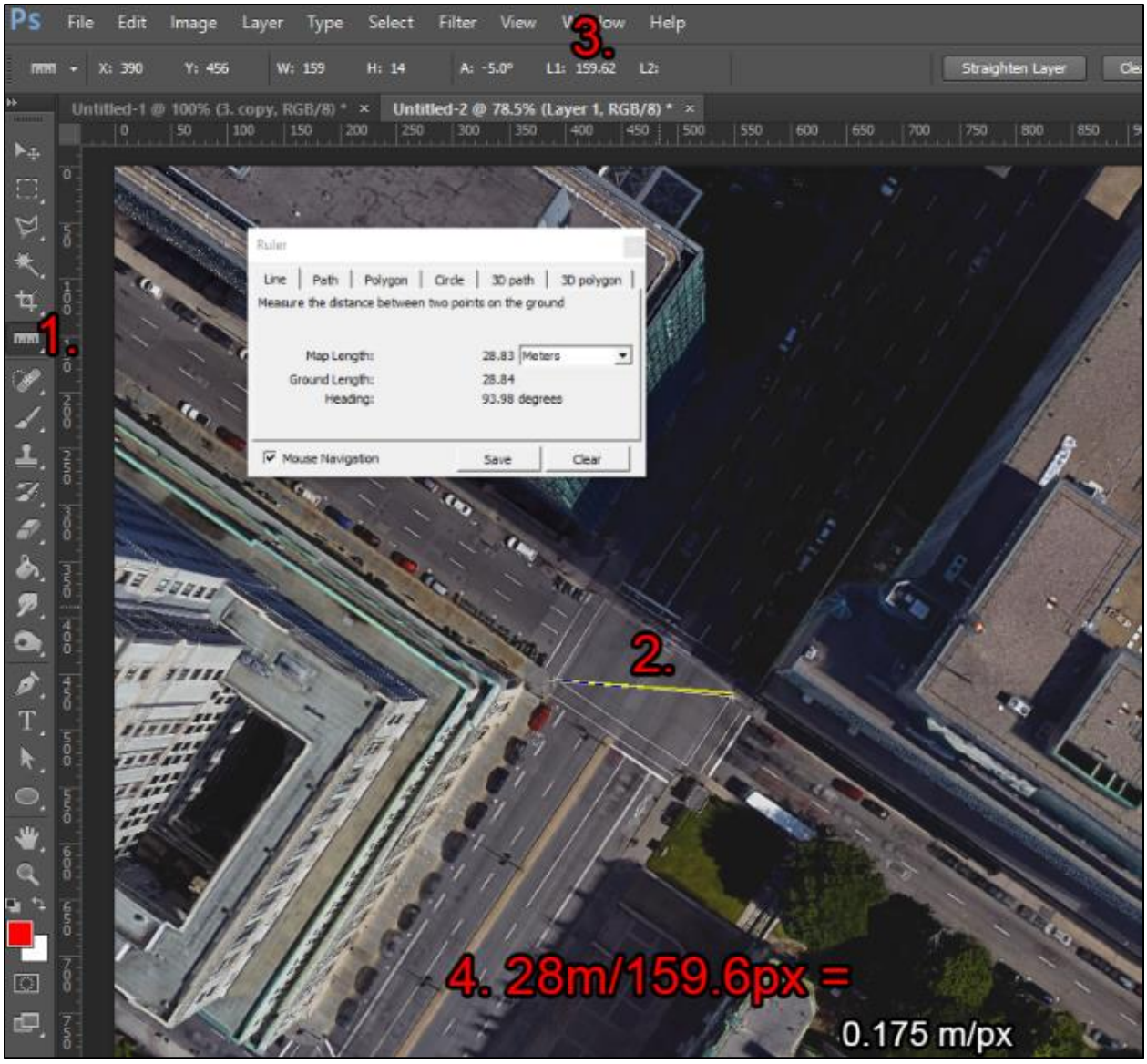
Each site folder should have a satellite image stored alongside with the site. By convention, this file is named “ortho-cal.png”, but may be named alternatively if so specified in the `scene.sqlite`.



The resolution of this image must be known to properly calibrate the homography later. To do so, divide a known spatial distance (e.g. as measured in Google Earth) with the measured distance in pixels (e.g. as measured in a photo editing program like Photoshop). It may by beneficial to hard-code this measurement into the image for easy reference.



TIP: In Google earth, be sure to align the camera such that it faces perpendicular to the ground, and faces north, by using the “R” key. Also, this image should be large enough to cover anything visible in any of the cameras installed at this site.



Video Sequences

All video sequences from the same camera are to be stored together in the same video folder.

Optional step: A still frame of one of the sequences needs to be captured for undistortion and homography calibration later. For best results, this frame should have a minimal amount of traffic present. The still should have the same filename as the sequences (plus an image file extension).

VLC is recommended for this task. Alternatively, tvaLib can do this automatically from the first video frame by running the following command from the tvaLib directory:

```
main.py -e --framedump
```

This should result in something similar to this:

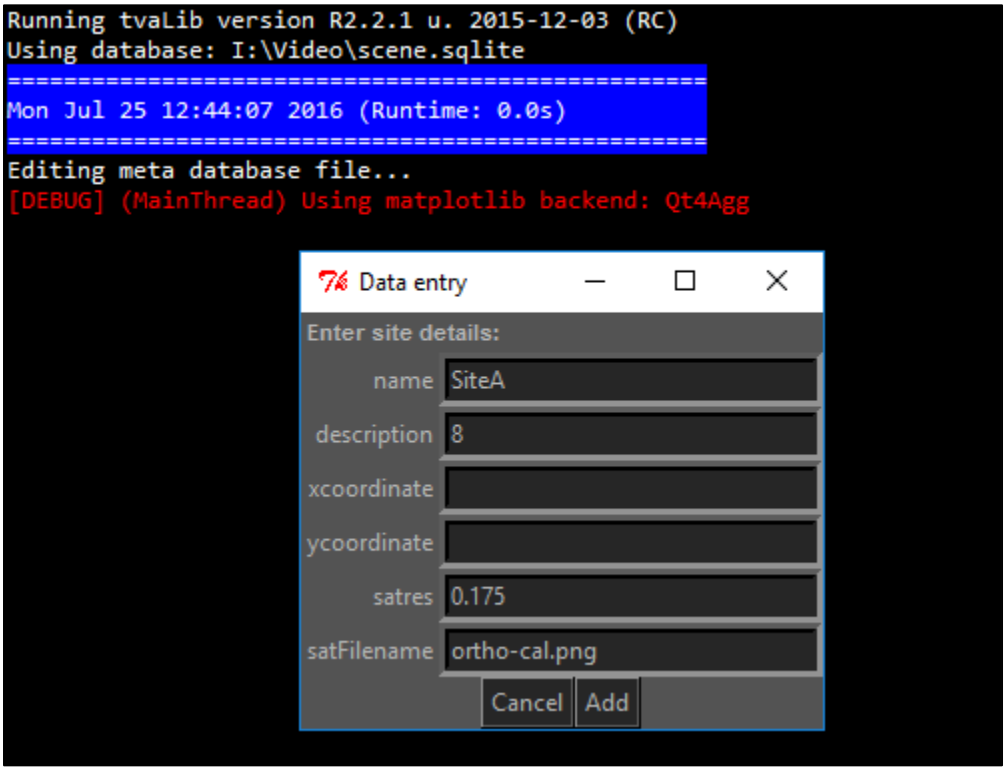
Video Database

The scene.sqlite needs to be edited to add indexes for all sequences. This can either be done manually using a SQLite database editor such as http://sqlitebrowser.org/, or using tools that come with tvaLib.

To add a new **site** to the database, run the following command from the tvaLib directory:

```
main.py --create-site
```

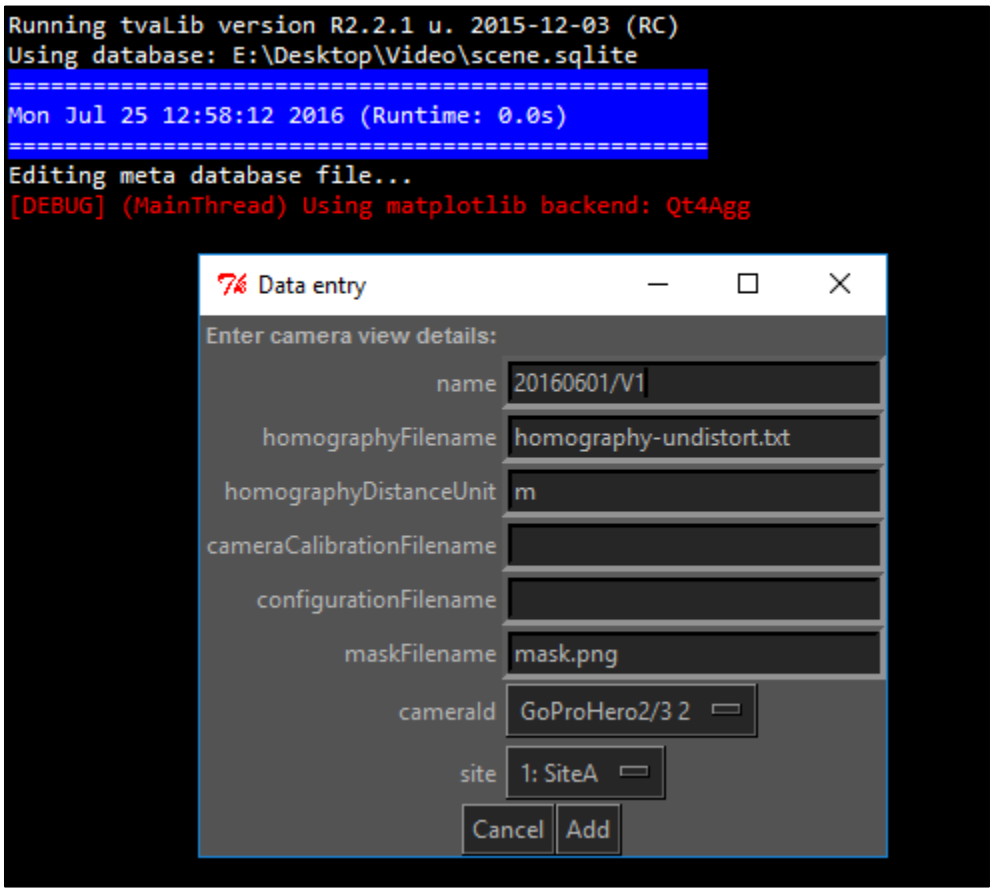
You will be prompted to enter the site's name (**this MUST be identical to the folder name used to store the data associated with this site**) an optional description (or database ID), the resolution of the satellite image calculated earlier, and the filename of this image ("ortho-cal.png" is the default).



To add a new **camera view** to the database, run the following command from the tvaLib directory:

```
main.py --create-view
```

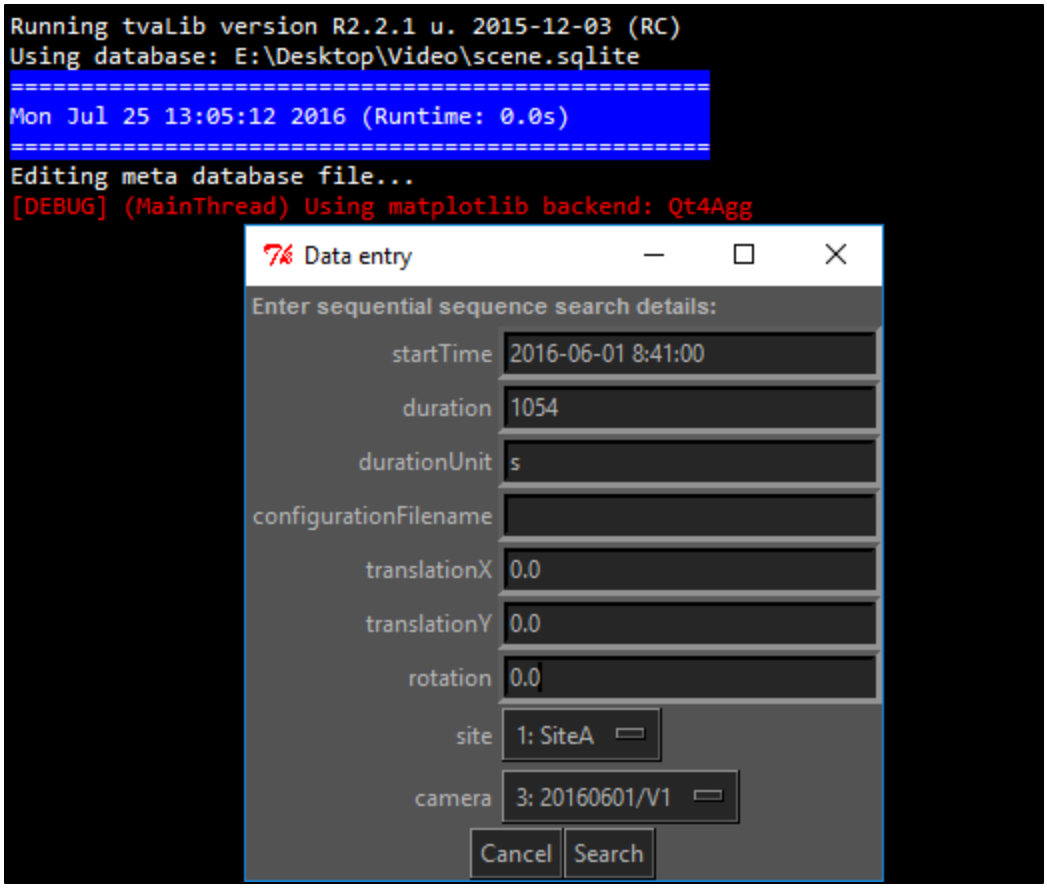
You will be prompted to enter the camera's name (**this MUST be identical to the folder names used to store the sequences associated with this camera view**), a camera type, and the relevant site at which recording took place. Note that in this case, sequences are stored in the "V1" folder within the "20160601" folder, itself within the "SiteA" folder. Thus the name for this camera becomes "20160601/V1".



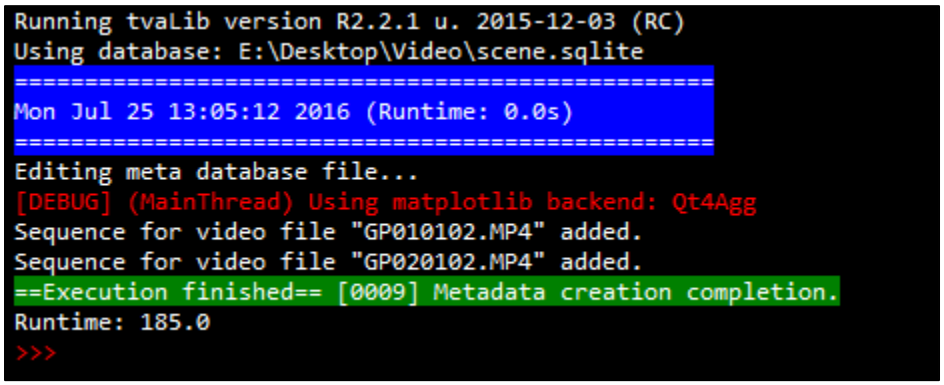
To add the sequences to the database, run the following command from the tvaLib directory:

```
main.py --create-seqs
```

You will be prompted to enter the start time (startTime) of the first sequence in the folder, and the duration of each sequence, in seconds. If the last sequence has a duration of a different length than the other sequences, this duration can later be edited manually in the database directly. The startTime takes strict ISO 8601 formatting https://en.wikipedia.org/wiki/ISO_8601 and should be entered with care.



This operation will search the folder for any video sequences, and add them to the database.



3.3. Preprocessing

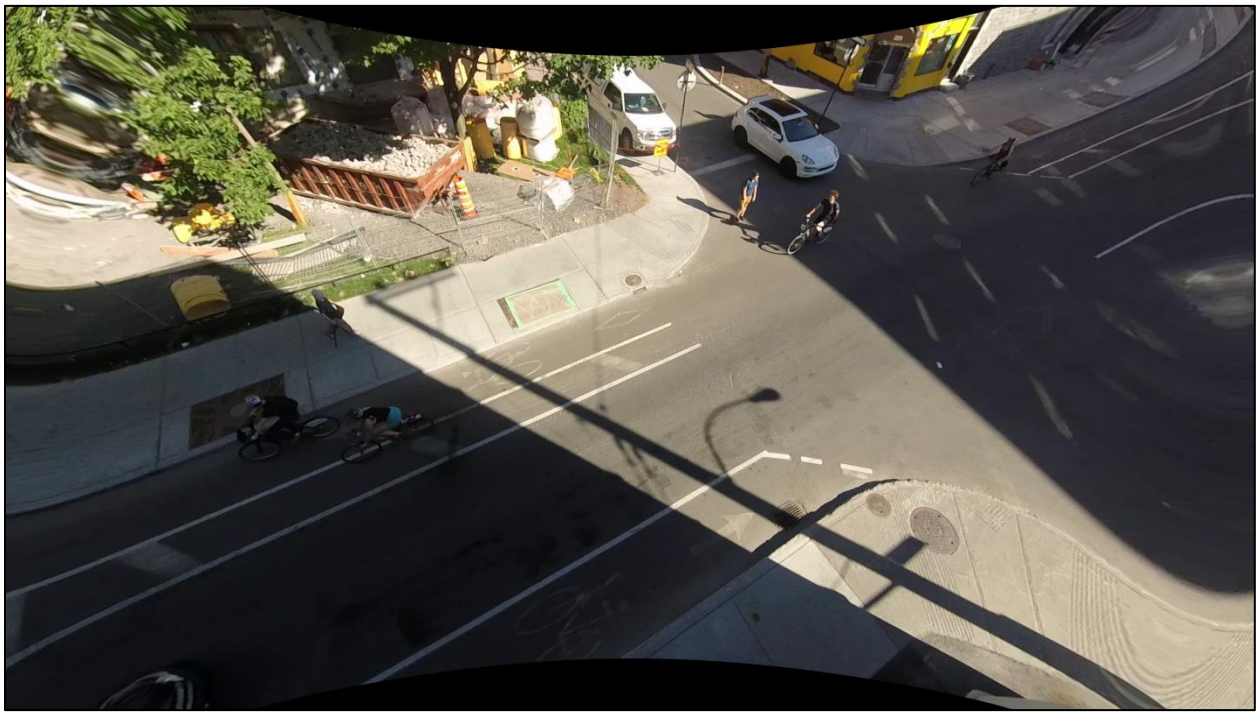
Undistortion

Once this is complete, an undistorted reference frame can be created, e.g. from the “GP010102.png” image captured in step 2.2.3 Video Sequences. The parameters should be properly set by having chosen the appropriate camera type in the previous section. To run the undistortion, run the following command from the tvaLib directory:

```
main.py -e --undistort
```

Follow the on-screen sequence selection dialogue and the program will then begin undistorting both the relevant still capture, and any relevant video files. Note that video undistorting is not necessary at this step, only undistortion of still frames. The program can be terminated as soon as the still frame has been undistorted.

```
Running tvaLib version R2.2.1 u. 2015-12-03 (RC)
Using database: E:\Desktop\Video\scene.sqlite
================================================
Mon Jul 25 13:37:54 2016 (Runtime: 0.0s)
================================================
Site listing:
 1: SiteA  (Cams: 20160601/V1)
Interactive site selection mode: choose a site (defaults to all):[DEBUG] (Mair
Thread) Using matplotlib backend: Qt4Agg
1
Alignments:
Cameras:
 1: 20160601/V1  [Ting's Calibration 1]
Interactive camera selection mode: choose a camera (defaults to all):1
Sequences:
 1: GP010102 [**NO DATA**]
 2: GP020102 [**NO DATA**]
Interactive sequence selection mode: choose a sequence (defaults to all):
================================================
Mon Jul 25 13:37:58 2016 (Runtime: 4.0s)
================================================
Undistortion for SiteA/20160601/V1/GP010102...
    Undistorting image...
    Undistorting video...
[>                                   0%                                   ]
[>                                   1%                                   ]
```



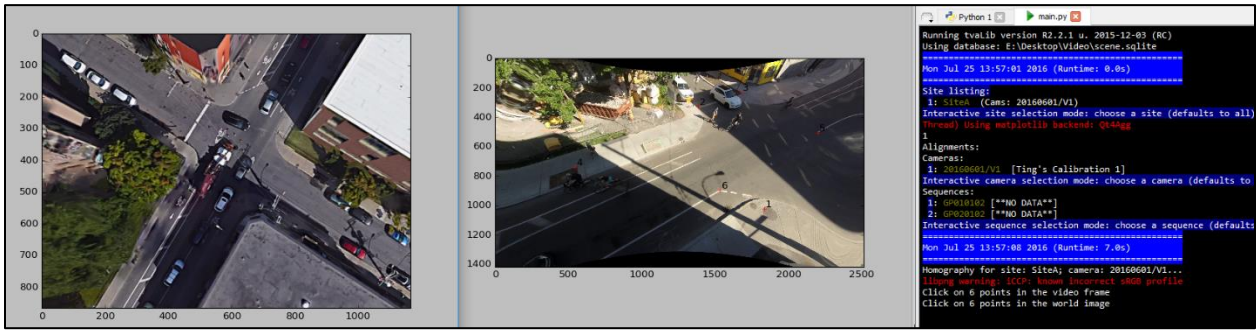
Note that if a still frame wasn't generated in a previous step earlier, one will now be automatically generated.

Homography

To calibrate the scene, a homography must be created for each camera view (or anytime the camera moves) to create the homography, run the following command from the tvaLib directory:

```
main.py -e --homo 6
```

The undistorted camera view will appear on screen and the user is prompted to select 6 distinct points. These points should be carefully chosen such that they be visible from both the satellite image and from the camera view, and should encompass the general area of interest of the camera view (e.g. well dispersed throughout the camera view). Once the selection of these 6 points is made, the user is prompted to select the same 6 points, in that order, in the satellite view.

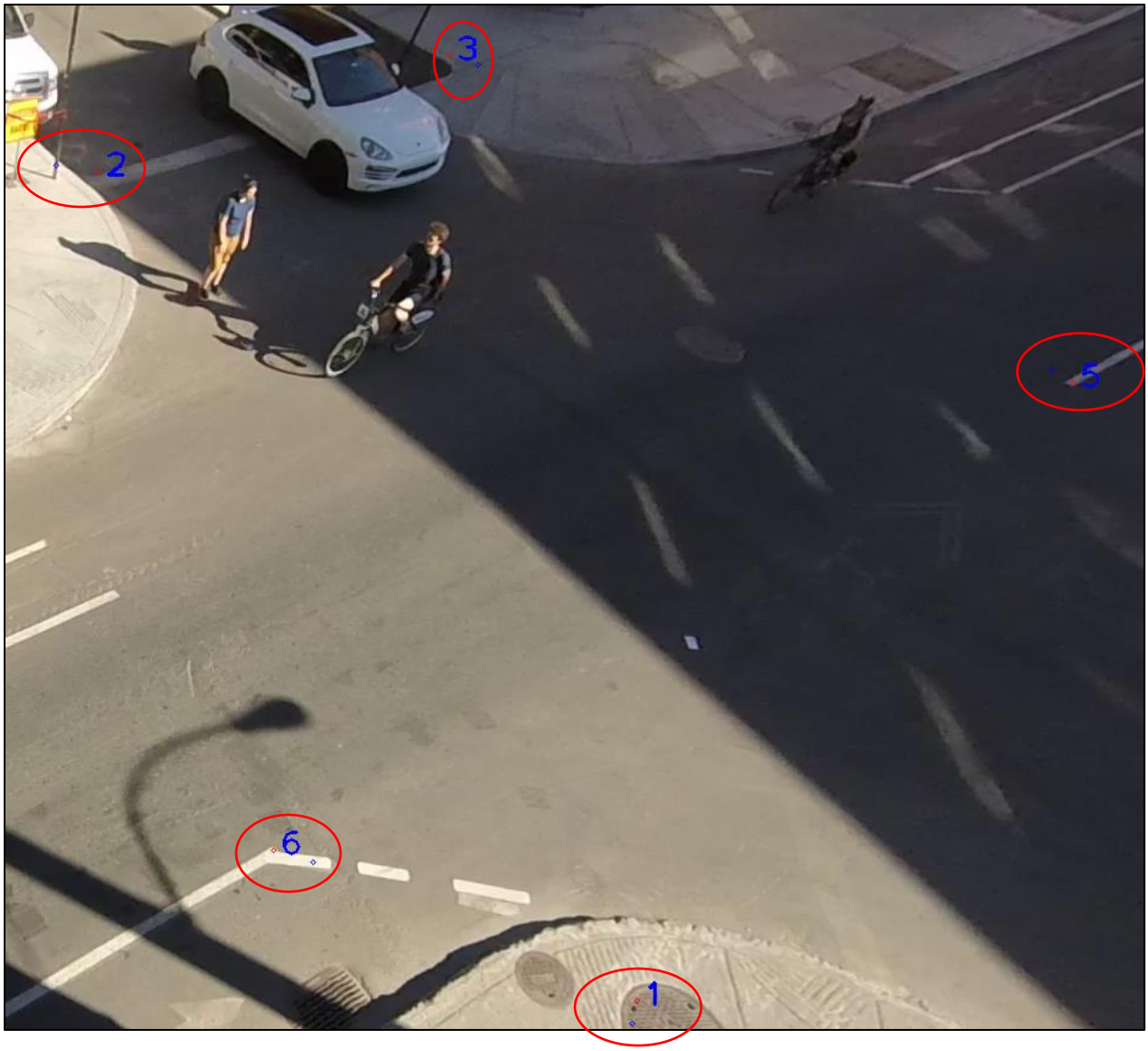


Once complete, the homography will be computed. Quality of the homography can be inspected visually by comparing how well the points line up in camera space. The homography is stored to a text file along with the sequences.

Note that if a still frame wasn't generated in a previous step earlier, one will now be automatically generated.

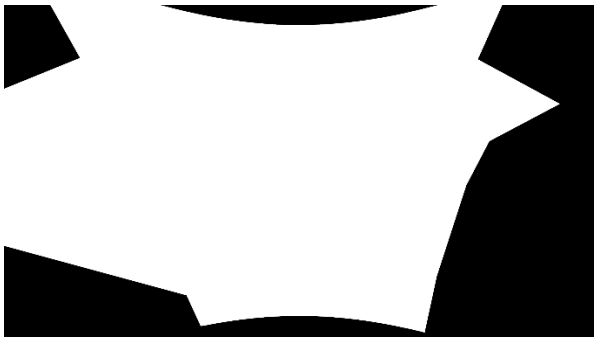
Usage Tips:

- Use at least six (6) correspondence points.
- Space the points out so that they cover the full scope of the image as much as possible – do not place correspondence points in the center of the image only.
- Avoid placing points in a line.
- Once the points are selected, verify that all correspondence points (blue and red points) match as closely as possible (roughly 10 cm).


3
2
5
6
1

Create Tracking Mask (Optional)

An optional step is to create a **tracking mask**, which defines an area of the image to ignore when running tracking using Traffic Intelligence. This image must have an identical size to the video and must be placed in the camera view folder alongside sequence and video files. The name of this mask is specified by `maskFilename` field entered during the camera view creation step outlined in section 3.2 (defaults to `mask.png`). At this time, the file should be created using an external image editing program. Any white pixels in the image correspond to the region of the image where Traffic Intelligence should attempt to track road users. Black pixels will be ignored.



The main benefit to this mask is that it improves performance and keeps trajectory file sizes down by ignoring portions of the video.

Usage Tips:

- The tracking mask is most beneficial for ignoring trees blowing in the wind.
- The tracking mask does not solve the issue of warm-up error, where partial objects are tracked at the edges of the image. Use a conventional (tvaLib) **mask** to solve this problem.
- If the video is being undistorted, **be sure to use the image dimensions of the undistorted video**. The easiest way to do this is to load a corresponding `*-frame-undistort.png` file after creating the homography.

4. Feature Tracking

To begin feature tracking, run the following command from the tvaLib directory:

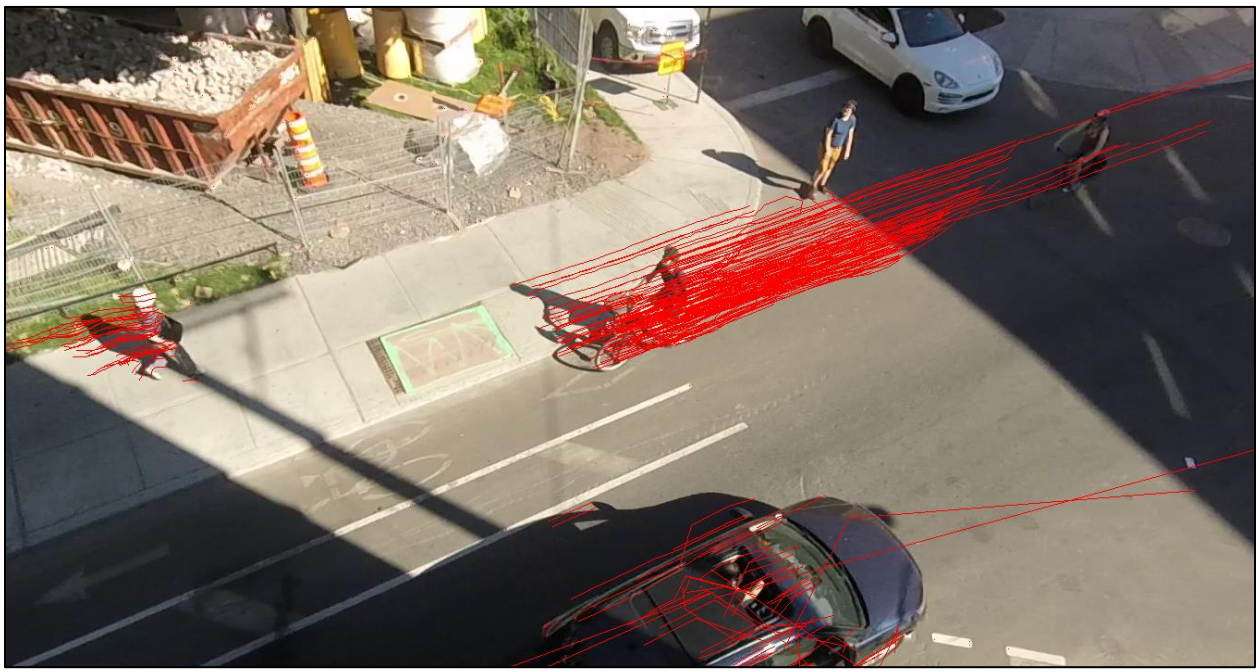
```
main.py -e --trafint
```

This will automatically launch feature tracking and grouping using Traffic Intelligence, with any appropriate homography transformations, tracking configurations (specified in the metadata), and any relevant undistortion.

To watch tracking, add the command `-p`. This will slow tracking down considerably, and is not recommended other than for manual inspection.

To process tracking on multiple sequences simultaneously using multiple CPU threads, use the command `-t #` replacing # with the desired number of parallel threads.

Tracking can take several hours to complete and should be monitored through task manager in Windows or the command `top -c` in Linux. These should also be used to terminate the tracking processes as needed.



5. Annotate Metadata

Once tracking is complete, annotation can begin. Some annotations are tied to specific sites, other annotations are tied to the specific study, to the camera, or to individual video sequences. These will be defined and created in this section

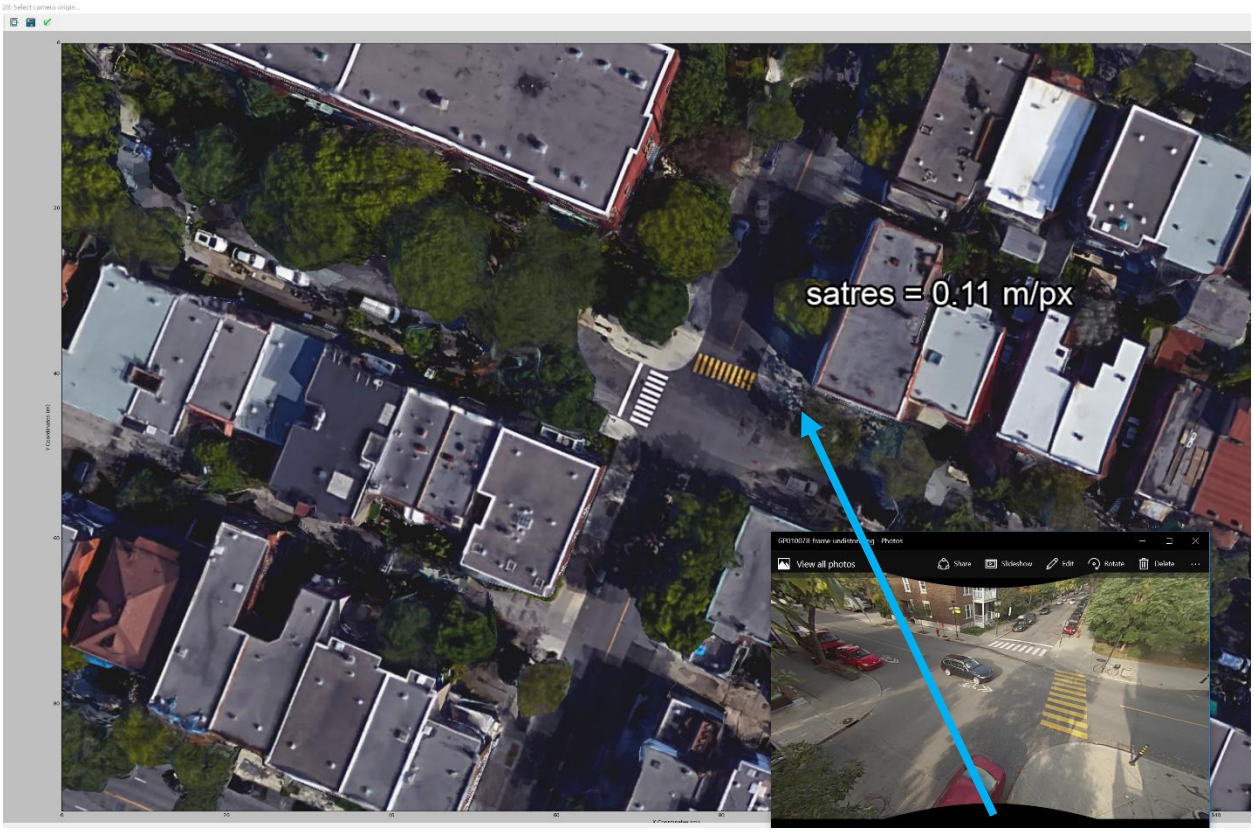
5.1. General Annotation

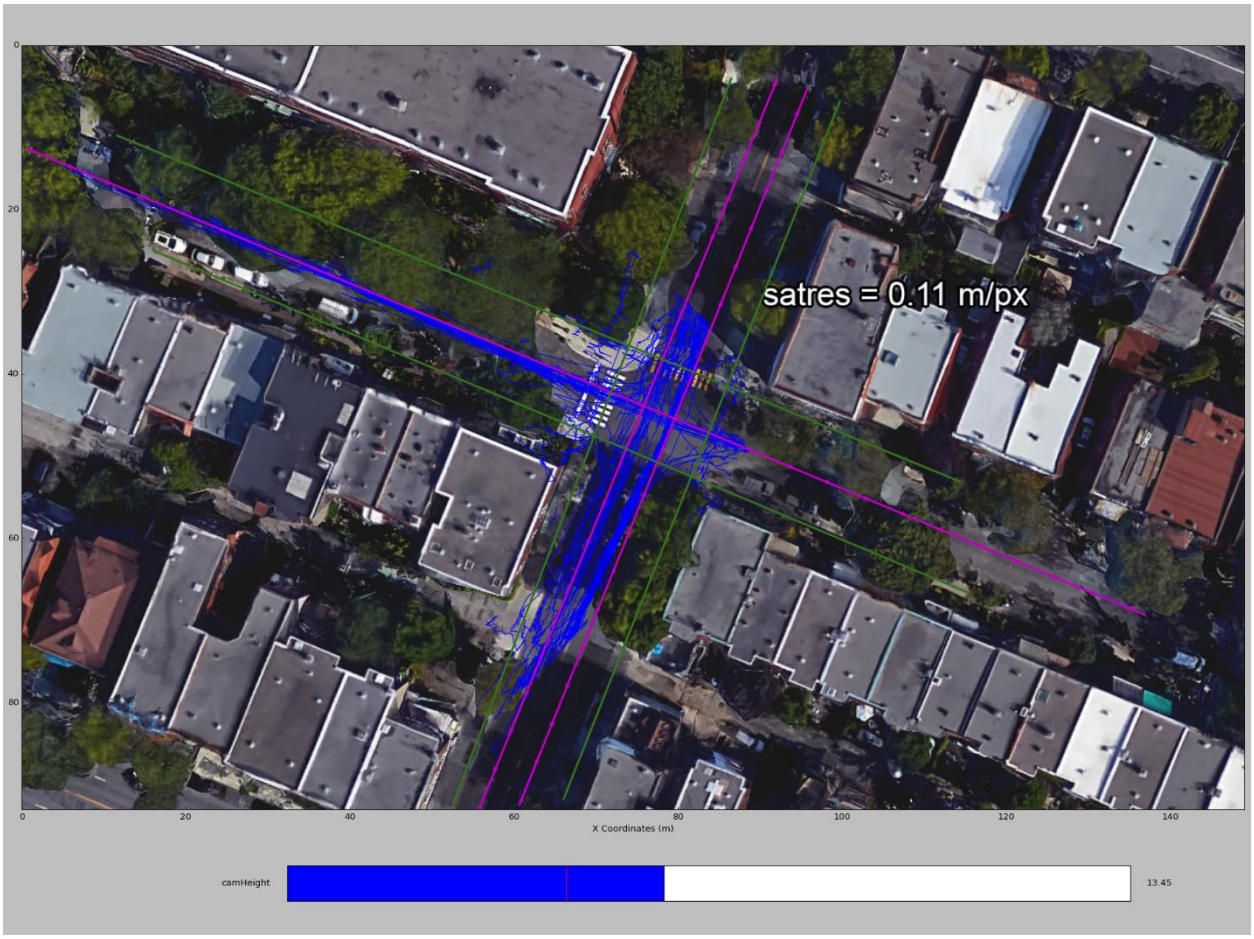
Correct Camera Parallax

Before analysis proceeds, it may be helpful to correct for camera parallax. This can be accomplished simply by annotating the known camera position and height. To do so, run the following command from the tvaLib directory:

```
main.py -e --draw-mhc
```

You will then be prompted to select the camera origin (if not already defined) and then you will be asked to calibrate the mast height (if not recorded in the field) by checking against a live correction of trajectory projection.



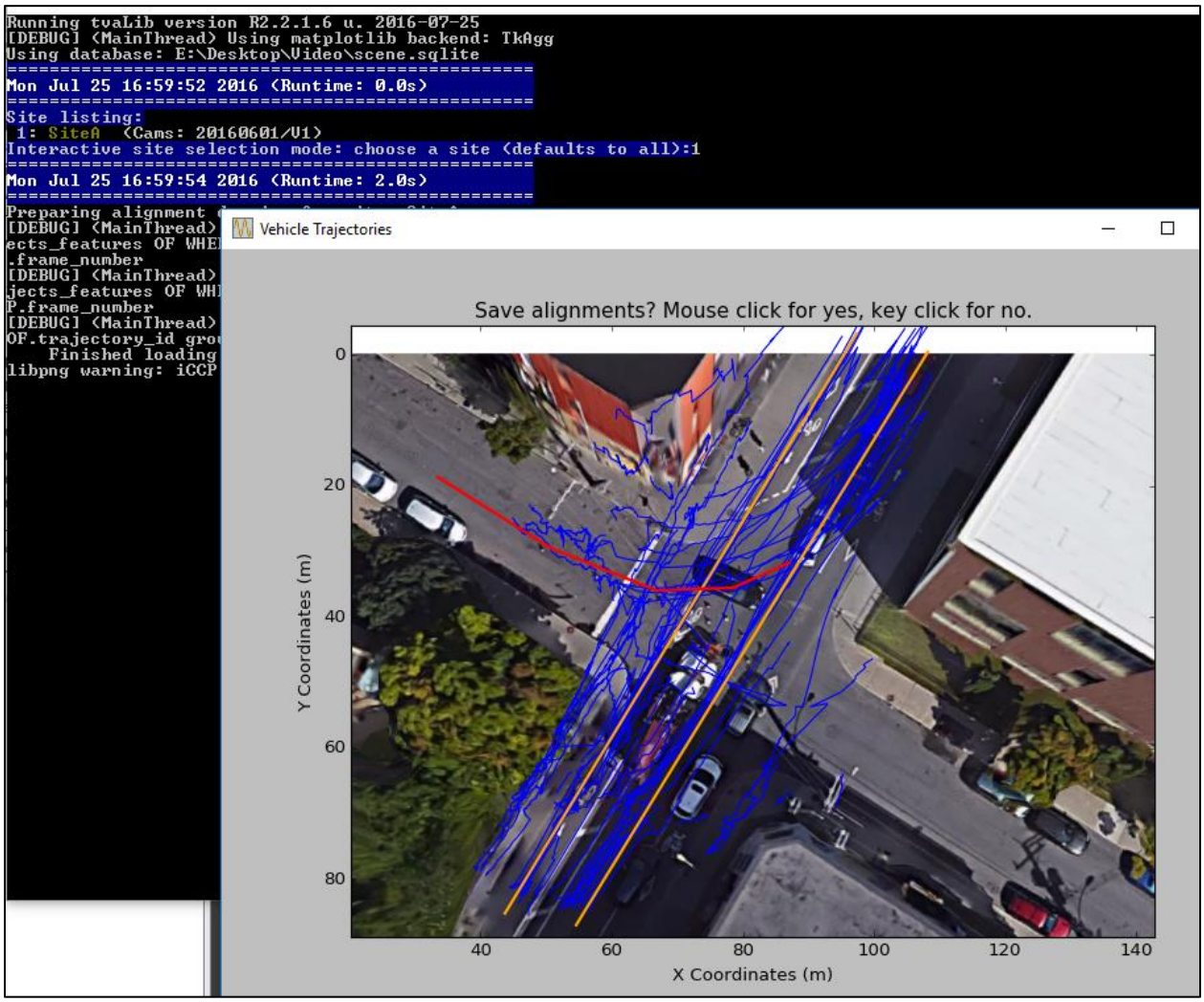

satres = 0.11 m/px
0
20
40
60
80
0
20
40
60
80
100
120
140
X Coordinates (m)
camHeight
13.45

Create the Alignments

The primary general-purpose annotation is the alignment. To annotate site alignments, run the following command from the tvaLib directory:

```
main.py -e --draw-align
```

The objective of this step is to draw center-lane alignments in the direction of travel of each lane. The alignment should ideally be representative of aspects of geometry (the lanes), and the clusters of trajectories of the tracked road users using these lanes. Generally, if the trajectories do not line up with the lanes (as seen in the satellite images), then there is likely a problem with the homography, or the angle of the field of view of the camera is so large as to create important parallax error with trajectories at a great distance from the camera.



The program will normally load a sample of trajectories for each camera attached to the site. Normally, a single set of alignments should serve all camera views at a particular site, and are created for all cameras simultaneously. To force tvaLib to display a drawing plot without any trajectories, running the following command from the tvaLib directory:

```
main.py -e --draw-align-no-trk
```

You will be prompted to verify automatic alignment connector creation (currently partially implemented) as well as identify pedestrian and bike paths (optional). Reffer to the following chart for alignment colour codes:

Alignment colour	Meaning
Red	Active alignment
Orange	Inactive newly drawn alignment
Magenta	Default/motor vehicle lane
Green	Sidewalk/crosswalk
Purple	Exclusive cycle lane

Usage Tips:

- Use **one alignment per lane, per direction**, as well as one alignment per sidewalk/crosswalk.
- Alignments should generally continue **straight through an intersection** whenever through traffic is expected (including sidewalks/crosswalks).

- **Do not draw connectors** between alignments. Try instead to draw them sufficiently close that the algorithm finds its own connections.
- When a car lane is shared with bikes, do not draw a separate alignment for bikes. **Bike lanes** should only be used when an area of road is **used clearly and exclusively for cyclists**.
- Alignments can cross each other perpendicularly, but should never overlap more than once.

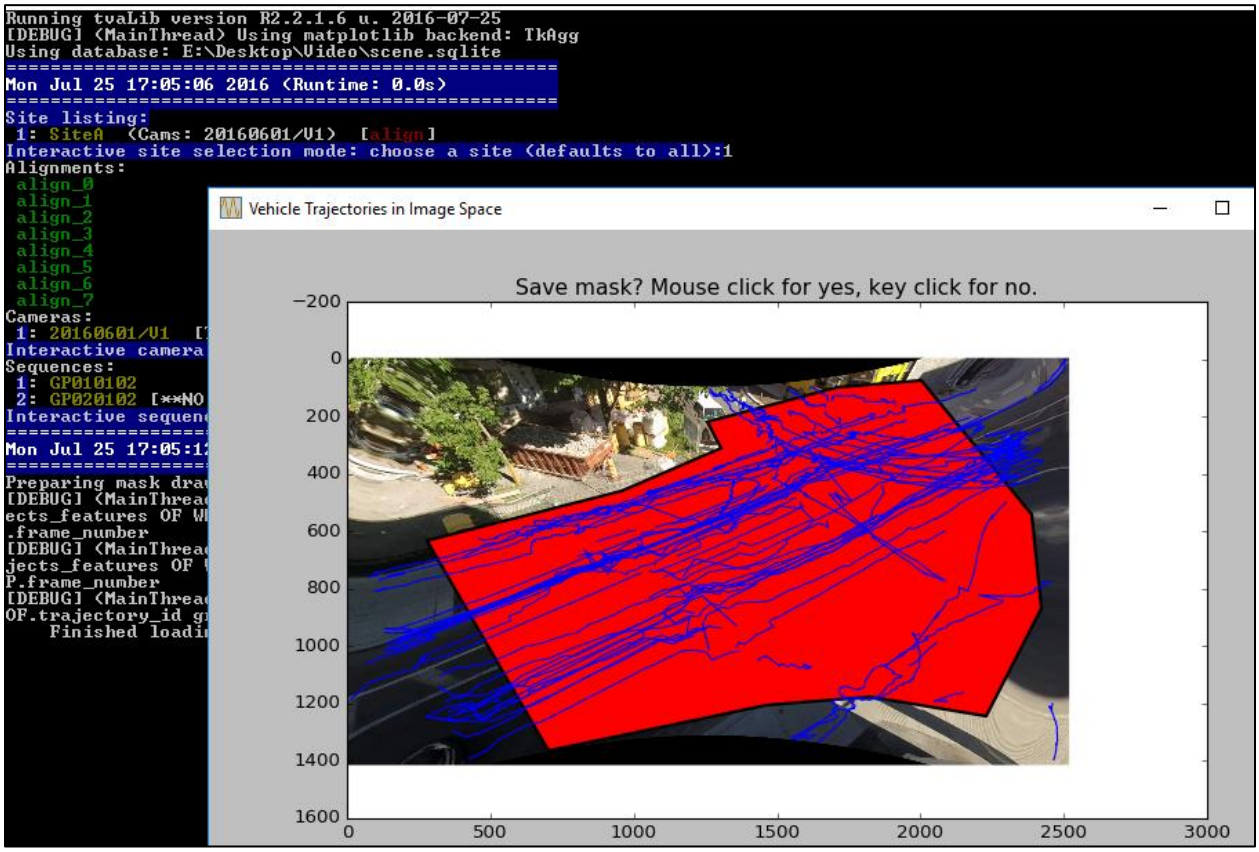
Create the Mask (Optional)

An optional step is to create a **mask**, which automatically rejects trajectories outside of a given area. This area is represented by the black polygons visible in the previous image. To draw the mask, run the following command from the tvaLib directory:

```
main.py -e --draw-mask
```

This same mask can be drawn using an alternate view by running the following command:

```
main.py -e --draw-mask2
```



5.2. Study-Specific Annotation

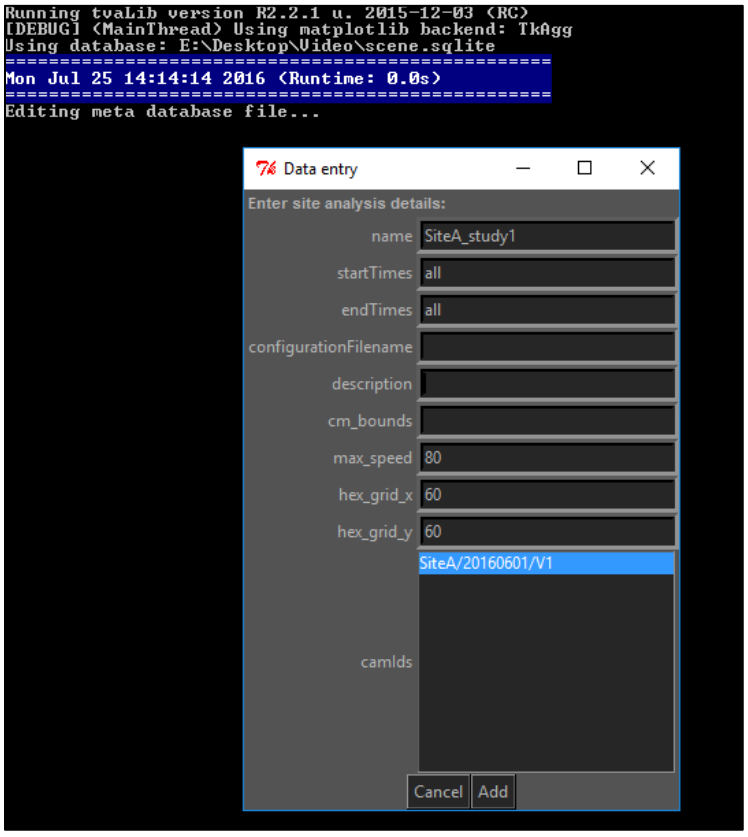
Create the Site-Analysis

The first step is to create a **site-analysis**, an object to store information relevant to the study of a particular site or camera view, or sequence.

To create a site-analysis to target a specific site, camera view, or sequence, run the following command from the tvaLib directory:

```
main.py --create-sa
```

As with previous database creation steps, this will launch a dialogue, asking for a name for the site-analysis, and a selection of cameras to include. Selection of trajectories by a period of time is currently not implemented.

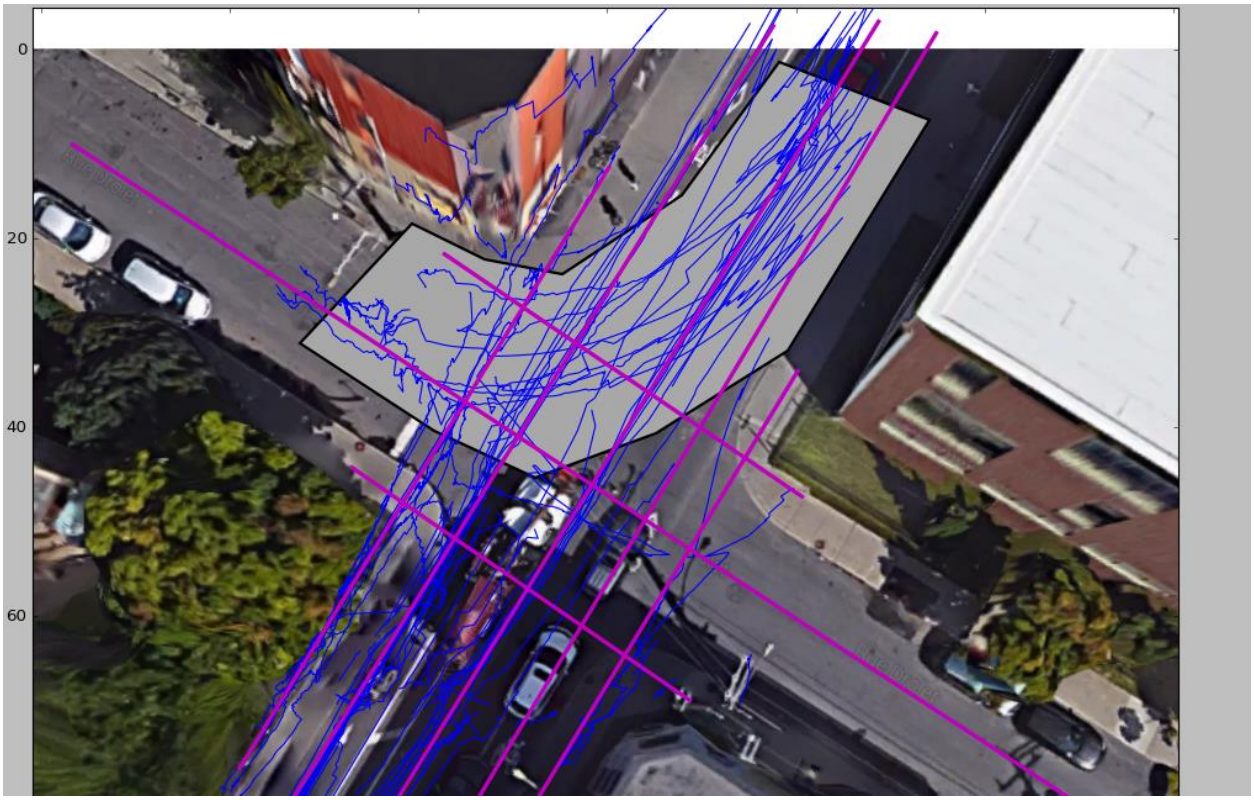


Draw the Analysis-Zone

The **analysis-zone** is a property of the site-analysis that defines a region in space (and time) where trajectories are selected for further analysis (hence site-*analysis*). To create an analysis-zone, run the following command from the tvaLib directory:

```
main.py -e --draw-zone
```

The location of the analysis will largely be dependant on the goals of the study. In the following figure, only road the left-turn area serving the south-east-heading alignment is selected for analysis.



Draw Plotting Bounds (Optional)

The **plotting bounds** method can be selected by running the following command from the tvaLib directory:

```
main.py -e --draw-bounds
```

This will zoom all figures (including all drawing figures mentioned in this section) to the selected world-space bounds. This data is mostly cosmetic.

5.3. High-Level Interpretation

HLI Modules extend tvaLib to add context-specific data to the analysis. These are usually study-specific modules. An example application of this is included with tvaLib source: a `stop` sign identification and analysis module. With this module, the user identifies a location along alignments were a stop sign is located, and tvaLib will automatically extract and compile the speed of road users at this stop sign.

To identify stop signs in trajectory space, the user must launch the following command:

```
main.py -e --draw-hli -y stop
```

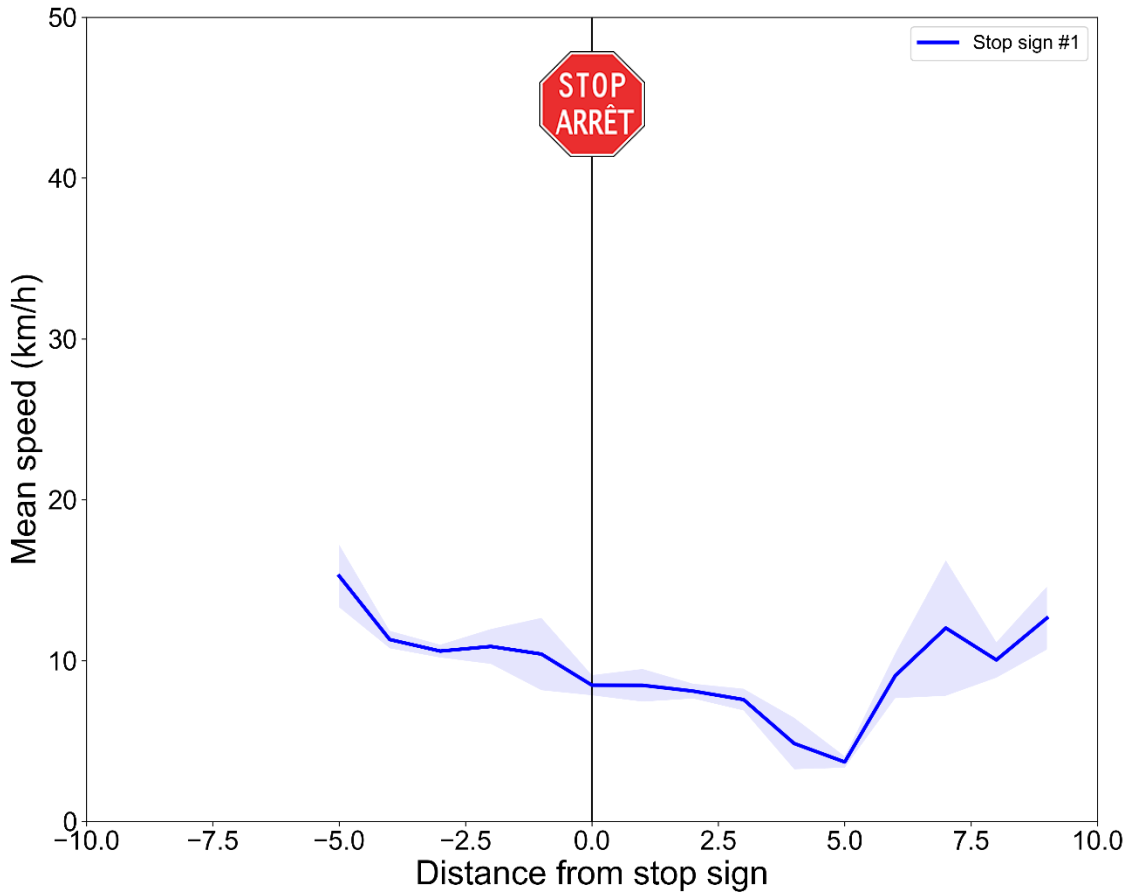
This opens up a simple interface where the user clicks on an alignment corresponding to the world location of the stop sign. Note that the `stop` argument identifies the `stop` HLI module by name. The argument `all` can also be used to run all available HLI modules.

Usage Tips:

- If the alignment data is changed for a site, the stop sign(s)'s location(s) will have to be reidentified for that site as well.

To perform the HLI analysis and return output, run the following command:

```
main.py -ewa -y stop
```



6. Analysis

6.1. Basic Traffic Analysis

A basic analysis can simply be performed with the following commands:

```
main.py -e
```

To save figures, add the `-w` option, as in:

```
main.py -ew
```

To also cache the data such that it loads faster during a future analysis, add the `-a` option, as in:

```
main.py -ewa
```

All program output is stored in the `Analysis` folder under the root video database folder, e.g. `I:\Video\Analysis`, with one folder per site-analysis.

6.2. Conflict/SSM/Interaction Analysis

Conflict analysis can be performed by launching the following command:

```
main.py -ea -i 0
```

Note that the `-i` parameter takes as input a list (e.g. `-i 0`, or `-i 0,1`) of prediction methods declared by id. The following methods are supported:

id	Prediction Method	Requires calibration
0	constant velocity	No
1	normal adaptation	No
5	discretised motion pattern	Yes

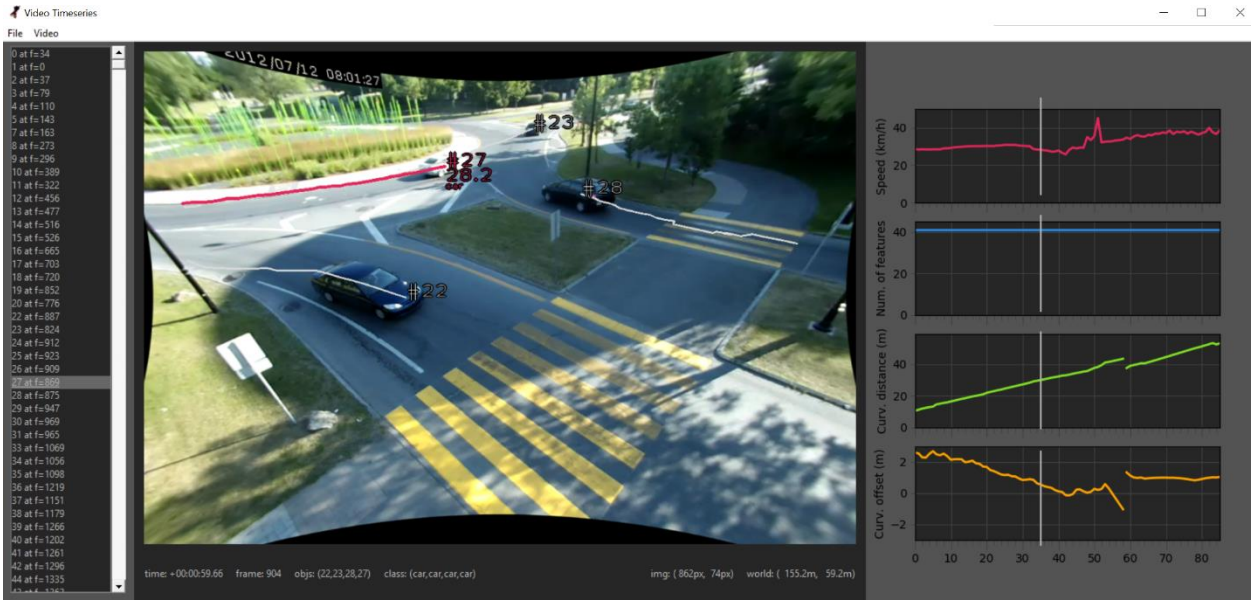
6.3. Playback

A video/trajectory playback interface is available to investigate individual trajectories or interactions manually, featuring video media controls along with tracking data. To launch video playback of trajectories, run the following command:

```
main.py -ep
```

To launch video playback of interactions (using constant velocity prediction, i.e. #0) and display collision points, run the following command:

```
main.py -e --play-int -i 0
```


Video Timeseries
File Video
0 at f=34
1 at f=0
2 at f=37
3 at f=79
4 at f=110
5 at f=143
7 at f=163
8 at f=273
9 at f=296
10 at f=389
11 at f=322
12 at f=456
13 at f=477
14 at f=516
15 at f=526
16 at f=665
17 at f=703
18 at f=720
19 at f=852
20 at f=776
22 at f=887
23 at f=824
24 at f=912
25 at f=923
26 at f=909
27 at f=869
28 at f=875
29 at f=947
30 at f=969
31 at f=965
33 at f=1069
34 at f=1056
35 at f=1098
36 at f=1219
37 at f=1151
38 at f=1179
39 at f=1266
40 at f=1202
41 at f=1261
42 at f=1296
44 at f=1335
#23
#27
#28
#22
time: +00:00:59.66 frame: 904 objs: (22,23,28,27) class: (car,car,car,car)
img: (862px, 74px) world: (155.2m, 59.2m)
Speed (km/h)
Num. of features
Curv. distance (m)
Curv. offset (m)
0 10 20 30 40 50 60 70 80

7. Validation and Manual Annotation

It is possible to validate the quality of the tracking using MOTP & MOTA evaluation between tracked trajectories and manually created ground truth data.

7.1. Annotation & Ground Truth Creation (tvaLib)

To create ground truth data, run the following command:

```
main.py -e --annotate
```

This will launch an interactive playback tool with annotation tools to add, delete, join objects and draw or edit existing trajectory keyframes. The most important keybord shortcuts:

Key	Function
Ctrl+N	Create new object starting at this instant
Ctrl+Left click	Jump to first frame of visible trajectory clicked on in the video window
Left click	Add or move existing keyframe at this instant
Delete	Delete any existing keyframe at this instant
Ctrl+Delete	Delete active object
Space Bar	Play/Pause
Arrow keys	Scrub forwards/backwards by one frame
Ctrl+ Arrow keys	Scrub forwards/backwards by ten frames
Page up	Select previous trajectory
Page down	Select next trajectory
4	Jump to previous keyframe of active object
6	Jump to next keyframe of active object
+	Speed up playback (x2)
-	Slow down playback (/2)
Home	Jump to first frame
End	Jump to last frame

Usage Tips:

- Keyframes should be added relatively frequently, e.g. **one keyframe approximately every second**. **More keyframes are needed when the trajectory accelerates or decelerates**. Do not forget to add a keyframe the instant a road user begins to move again.
- The selected position should correspond to the **center of the object where it touches the ground** in real space, as best as possible.
- The position of the object from one keyframe to the next should be consistent; **try to identify a feature/corner to follow**. If the object rotates or changes shape (e.g. pedestrians), this position will have to be estimated.

7.2. Annotation & Ground Truth Creation (Urban Tracker)

The Urban Tracker Annotation Tool can also be used to generate ground truth data. This tool is focused on creating bounding boxes around pixels. It is not as efficient for the task of evaluating MOTA as the

included tvaLib annotation tool. Urban tracker is available here:
https://www.jpjodoin.com/urbantracker/index.htm

7.3. MOTP & MOTA Analysis

With these annotations saved and exported to a ground truth file, you may evaluate MOTP & MOTA by running the following command:

```
main.py -e --mot
```

If you add the play command, as in:

```
main.py -ep --mot
```

tvaLib will also load both the ground truth data and tracking data for visually comparison using the standard playback interface.

7.4. MOTP & MOTA Optimisation

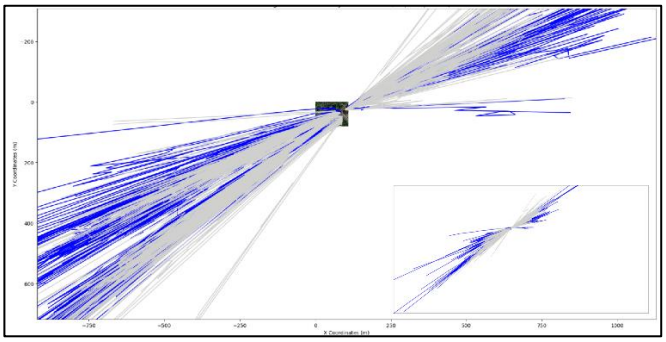
You can also attempt an optimisation of tracking parameters using the ground truth data with the following command (using multiple threads is highly recommended):

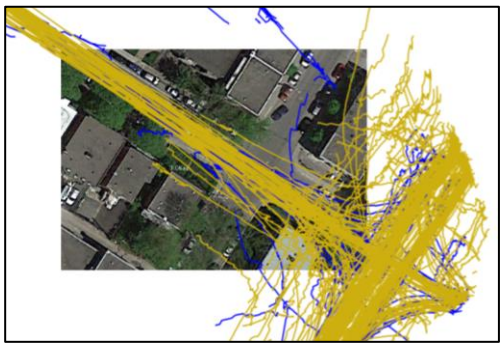
```
main.py -ep -t 8 --mota-opt-t --mota-opt-g
```

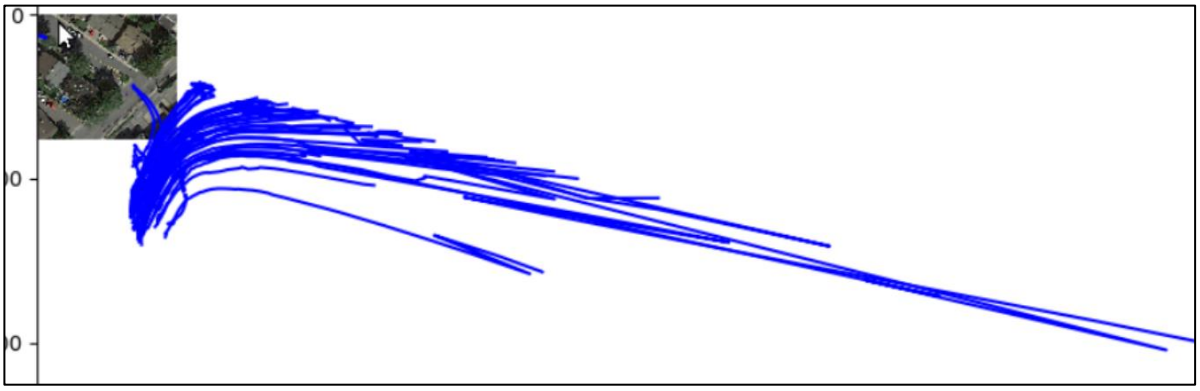
8. Troubleshooting

8.1. Missing or erroneous homography

Problems with homographies usually result in poor trajectories relative to the environment. The following examples are symptomatic of poor camera calibration.







Trouble shooting steps

Image clearly not centered: wrong camera resolution set in camera settings

1. Before creating the homography, confirm visually that undistortion settings are correct. In particular, check for warping at image edges, and verify that the image is centered.
2. Redraw the homography as per the instructions in *Section 3.3 – Homography*.
    a. Use at least six correspondence points.
    b. Space the points out so that they cover the full scope of the image as much as possible – **do not place correspondence points in the center of the image only.**
    c. Avoid placing points in a line.
    d. Once the points are selected, verify that all correspondence points (blue and red points) match as closely as possible (roughly 10 cm).

9. Programming Guide

9.1. HLI Plugins

tvaLib supports modularity of analysis by using plugins called *High-Level-Interpretation* (HLI). HLI allows the user to implement their own analysis, computation, and visualisation functions and have them run in-line with tvaLib. This is particularly useful when custom, analysis-specific calculations need to be made, or when the user wishes to extend general-purpose analysis functionality of tvaLib.

A tvaLib HLI plugin is a self-contained Python library file that is placed in the tvaLib `hli` folder. A tvaLib HLI plugin contains its own function and class definitions alongside several mandatory, reserved functions that are called by tvaLib. Thus, as a library, **it should not be written to execute code on its own**.

Usage Tips:

- With the exception of imports and simple variable declarations, no code within an HLI module should be outside of a class or function definition.

Here are the following reserved functions and what they do if the associated HLI module is specified by name:

Function Name	Arguments	Purpose
main()	commands, config, objects, sites, site_analyses	main() is executed at the end of the main() function of tvaLib (during normal operation). It loads all relevant user commands, configuration settings, all objects, and all metadata. This function should conduct the bulk of your analysis, to be performed for each site individually by loading the relevant metadata.
listingPlugin()	site_analysis, config	This function is called by master interactiveSiteAnalysisSelection() to output metadata completion information during interactive site selection.
draw()	objects, commands, config, sites, site_analyses, local	This is called when the user specifies drawing of metadata (that may be necessary for performing HLI-specific calculations.
analysis()	commands, config, site_analyses, analyses, local	This is called when the analysis script is performed, usually to load precalculated HLI results and compile them together.

An example file, `example.template`, is included in tvaLib (be sure to rename it, e.g. `myHLItest.py`).

9.2. Trajectory Data

As with Traffic Intelligence, tvaLib uses the data structure provided by the Trajectory Management library. Each trajectory is a `MovingObject()` object with a number of methods that can be called to access the data. Here is a simple (inefficient) example that can be run from within an HLI's function that searches for the closest trajectory vertex to the given point at coordinates (80, 70) for the first object to appear after a time of 500 frames, returning the speed of that object at that point:

```
1  import lib.tools       as tvaLib
2  import math            as m
3
4  try: objects = objects.getAll()
5  except: pass
6
7  pos_x = 80.0
8  pos_y = 70.0
9  search_time_in_frames = 500
10
11 for obj in objects:
12    if(obj.getLastInstant() < search_time_in_frames): continue
13    nearest_dist = sys.maxint
14    nearest_point = 0
15    for pIx in range(len(obj.getXCoordinates())):
16       dist = tvaLib.Geo.ppd(pos_x,pos_y,obj.getXCoordinates()[pIx],
                               obj.getYCoordinates()[pIx])
17       if(dist < nearest_dist):
18          nearest_dist = dist
19          nearest_point = pIx
20
21    speed = m.sqrt((obj.velocities.getXCoordinates()[nearest_point])**2+
22                   (obj.velocities.getYCoordinates()[nearest_point])**2)
23    print 'Nearest distance: ' + str(nearest_dist) + ' at point: ' +
            str(nearest_point)
24    print 'Speed: ' + str(speed)
25    break
```

Usage Tips:

- To get this code working right away, paste lines 4 to 25 into `hli/example.template` (on line 67, fixing the indentation as necessary), rename the file to `example.py`, and, finally, launch `main.py` with the arguments `-e -y example`.

Here is the code explained, line by line.

```
import lib.tools       as tvaLib
import math            as m
```

This line imports tvaLib's library of tools and basic Python math functions. The geometry-calculating point-to-point distance function `ppd()` will be needed later for the task. Normally, imports should be declared at the top of the file (you can find these lines of code in `example.template`), outside of any functions, but they can also be imported within a function at runtime, so long as the import happens before any other references to it.

```
try: objects = objects.getAll()
except: pass
```

This is a python try-except block. It will attempt to run thy code under the `try:` statement. If it fails due to some error, it'll be instructed instead to `pass` (i.e. ignore) execution of that code instead. `objects.getAll()` is a tvaLib-specific instruction that is beyond the scope of this example, but is necessary to declare here to get the example working in an HLI script. In any other context, it'll fail, but that isn't a problem since failure of this line of code is ignored.

```
pos_x = 80.0
pos_y = 70.0
search_time_in_frames = 500
```

These three lines store, in memory, some data that will be used as search criteria: coordinates for the search location and the minimum frame number to perform the search.

```
for obj in objects:
```

This is a basic python loop. For each trajectory (`obj`) in the list of trajectories (`objects`), the remaining indented lines of code will be run (unless a `break` is found). In this way, the computer is instructed to execute the remaining indented code for *each* trajectory, in the order that those trajectories are stored in memory.

```
if(obj.getLastInstant() < search_time_in_frames): continue
```

This is a basic python if statement. It evaluates if `obj.getLastInstant()` is smaller than `search_time_in_frames`. If it is, the next statement, `continue` is executed. Otherwise, it is not executed. Normally, the code would continue to execute after the `if` statement and any conditional code executes. However, in this case, `continue` is a special instruction which instructs the computer to ignore the remainder of the code to be executed in this iteration of the `for` loop and to, instead, execute the *next* iteration (trajectory) of the for loop immediately.

The value of `search_time_in_frames` was previously set to 500 and will stay constant for the remainder of this task. Meanwhile, `obj.getLastInstant()` accesses the last frame number of the current trajectory.

In summary, this lines checks that the current trajectory exists at least up until frame 500 before proceeding with any further calculations. If not, it instructs the program to skip any remaining calculations and jump to the next trajectory instead.

```
nearest_dist = sys.maxint
nearest_point = 0
```

Here, memory is set aside to remember which point was found to be closest and what that distance was. The initial distance is set to `sys.maxint`, which instructs the computer to use the largest number it is capable of remembering.

```
for pIx in range(len(obj.getXCoordinates())):
```

This is a second loop that loops through the index (stored locally in memory as `pIx`) of each trajectory vertex. For example, a trajectory with three vertices would result in `range(len(obj.getXCoordinates()))` having an iterable value of `[0,1,2]` (a Python list). Thus, this loop will run three times, running through the indices 0 to 2, corresponding to each of the trajectory's three vertices. Note that this `for` loop is nested in the *trajectory* **for** loop, i.e. calculations are being performing on *each* vertex of *each* trajectory…

```
dist = tvaLib.Geo.ppd(pos_x,pos_y,obj.getXCoordinates()[pIx],
obj.getYCoordinates()[pIx])
```

This instruction (one line of code) stores in memory (`dist`) the distance calculated by the function `ppd()` located in `tvaLib.Geo` which was previously imported. `ppd()` takes four parameters: the x and y coordinates of a first point and the x and y coordinates of a second point. It returns the distance between these two points. These coordinates are past be referring to `pos_x` and `pos_y` defined in memory earlier, as well as passing the indexed point of the list of X and Y coordinates. `obj.getXCoordinates()` returns a Python list of x coordinates representing the coordinates of the vertices of the `obj` trajectory. The specific vertex chosen corresponds to the index `pIx` of the current iteration through the vertex indeces.

```
if(dist < nearest_dist):
    nearest_dist = dist
    nearest_point = pIx
```

Here, the distance is compared to the existing shortest distance. If the distance of this vertex is smaller than the last, it becomes the new shortest distance. This is the last operation performed for each iteration of the vertex loop.

```
speed =
m.sqrt((obj.velocities.getXCoordinates()[nearest_point])**2+(obj.veloc
ities.getYCoordinates()[nearest_point])**2)
```

This instruction stores in memory (`speed`) the normalised speed value at the point that was found to be nearest (shortest `ppd()`). Trajectory speed is stored as vertices, as with positions, and is accessed in a similar manner.

```
print 'Nearest distance: ' + str(nearest_dist) + ' at point: ' +
str(nearest_point)
print 'Speed: ' + str(speed)
```

This line of code instructs the computer to print to console the results of the search and distance.

```
break
```

This line instructs the computer to stop executing any more iterations of the loop. In summary, the script stops searching through trajectories as soon as it finds a trajectory that exists after 500 frames.