OpenCV2X-Documentation

1. OpenCV2X With Veins:

This section includes all the documentation for the Veins based OpenCV2X implementation.

1.1 Installation:

1.1.1 Requirements:

- Ubuntu (16.04 and greater)
- SUMO 1 or greater: https://sumo.dlr.de/docs/Downloads.html
- GNU GCC 7.3
- OMNeT++ 5.4 or greater: https://omnetpp.org/download/
- Veins 5.0 or latest: http://veins.car2x.org/download/
- INET v3.6.6: https://github.com/inet-framework/inet/releases/tag/v3.6.6

Note: Different operating systems and versions as opposed to the above may in fact work but these are the versions which the model is known to work on.

1.1.2 Configuring the OMNeT++:

- Follow the usual procedure for OMNeT++ install https://doc.omnetpp.org/omnetpp/InstallGuide.pdf
- Importantly there is an issue with the osgEarth which will lead to issues with running simulations. I have no need for this and as such simply turn it off. This is done when going to configure OMNeT++ before making it you use

the below command this turns off the osgEarth functionality.

```
./configure WITH_OSGEARTH=no WITH_OSG=no
```

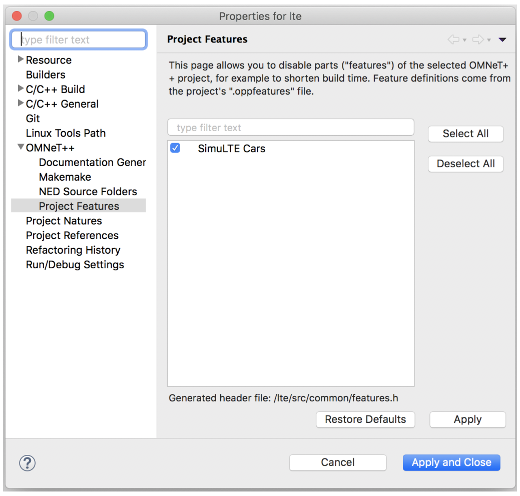
1.1.3 Install components:

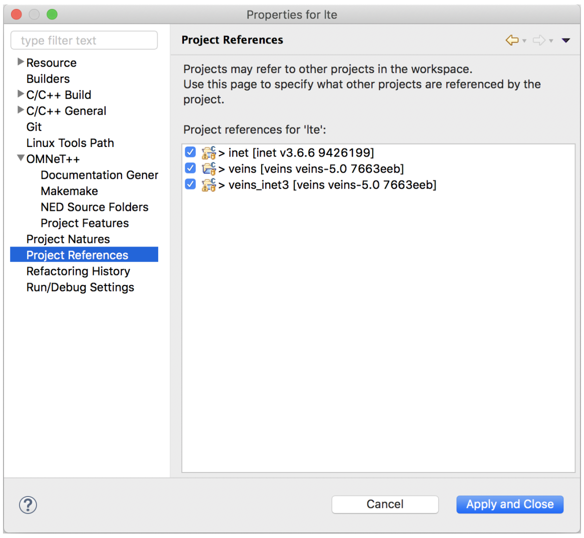
- Open the OMNeT++ IDE with the command: `omnetpp`
- Create a new workspace under your preferred name, ensure not to install **INET** through the dialogue when making a new workspace.
- Import the projects required, through the dialogues. `File | Import | General | Existing Projects into Workspace`
- Using this dialogue install both INET and SimuLTE (as the **lte** project) into the workspace.
- Finally add the veins project, but specifically for this one you want to also check the box for `Search for nested project`. Then you must install the **veins** project and the **veins_inet3** project

1.1.4 Project Features and References

Now the project needs to have the correct Project features activated to ensure SimuLTE runs correctly. Right-click on the **lte** project and select **Properties**

Ensure that the `OMNeT++ | Project Features` and the `Project References` are all ticked like the below images. i.e. **SimuLTE Cars** is active and **inet**, **veins**, **veins_inet3**.





1.1.5 Run Scenario

At this point you should be able to run a simulation, this is done the same way as running any veins based simulations.

- cd into the directory where veins is installed on your system and start the sumo launcher.

```
python2 sumo-launchd.py
```

- At this point you must go back to OMNeT++ and open the `omnetpp.ini` file in `lte | simulations | Mode4`
- Then you can simply click the `Run` button in the top left and a configuration will be made that will run your simulation. Though everything will build first and this can take some time but will be faster on subsequent runs.
- You can change the way these runs are done by going into the configurations through the drop down button next to the run button and selecting `Run Configurations ...`

1.2 Adding a new scenario Veins

Adding a new scenario for the veins version of OpenCV2X is similar to the approach for adding a scenario in veins itself or OMNeT++

- The simplest approach is to copy the Mode4 directory and then edit the necessary files to what you require for your own simulations.
- If you wish to implement something more complex than the basic application layer provided in the default scenario then you simply need to extend from the `Mode4BaseApp` which can be found in `simulte/src/apps/mode4App` this can be extended to implement whatever it is you wish to simulate.

1.3 Debugging Veins simulations

- Veins simulations are debugged using OMNeT++ with it's built in features which can be seen on the official OMNeT++ tutorial.
- What is worth highlighting is the fact that **SimuLTE** has to be configured in debug mode or release mode when changing between the two.
- This is done simply by right clicking on the SimuLTE project in the OMNeT++ IDE and under `Build Configurations->Set Active` select `gcc-release` or `gcc-debug`. See the image below

2. OpenCV2X With Artery:

This section includes all the documentation for the Artery based OpenCV2X implementation.

2.1 Installation:

2.1.1 Requirements:

- Ubuntu (16.04 and greater)
- SUMO 1 or greater: https://sumo.dlr.de/docs/Downloads.html
- CMake 3.13 or greater
- GNU GCC 7.3
- OMNeT++ 5.4 or greater: https://omnetpp.org/download/
- Boost 1.65.1

Note: Different operating systems and versions as opposed to the above may in fact work but these are the versions which the model is known to work on.

2.1.2 Configuring the OMNeT++:

- Follow the usual procedure for OMNeT++ install https://doc.omnetpp.org/omnetpp/InstallGuide.pdf
- Importantly there is an issue with the osgEarth which will lead to issues with running simulations. I have no need for this and as such simply turn it off. This is done when going to configure OMNeT++ before making it you use the below command this turns off the osgEarth functionality.

```
./configure WITH_OSGEARTH=no WITH_OSG=no
```

2.1.3 Make Components:

You must firstly make all the necessary submodules. This can be done through the `make all` command in the root directory of the project. This works for all dependencies except for veins which requires the following to configure and make:

```
cd extern/veins & ./configure & make
```

These can also be built separately by calling `make inet`, `make vanetza` and `make simulte`

This can often be preferable as running them in parallel is much quicker (though **INET** must be made before **SimuLTE**).

2.1.4 Cmake Setup:

Now you will need to setup a build direrctory for the project, as **Artery** and this project use **CMake** as its build system

```
mdkir build
cd build
cmake ..
```

Before you can fully build the system you need to ensure that the `WITH_SIMULTE` option is set in cmake. This is done in the build directory by using `cmake-gui` or `ccmake` if that is installed.

Now if you run `cmake --build .` The project will be built for CMake.

2.1.5 Run a Scenario:

Now you should be able to run the model using the following command.

```
cmake --build build --target run_simulte-cars
```

This will run the Open-CV2X highway-fast scenario.

2.2 Adding a scenario:

When adding a new scenario in Artery you need to edit the cmake project to point at your new scenario this is done by following the below.

- You must first edit the `CMakeLists.txt` file found in `path-to-project/OpenCV2X/scenarios/`
- Under the code section `if(TARGET lte)` you need to use the `add_opp_run()` command to add a new scenario
- The following is an example of what this command should contain.

  ```
  add_opp_run(scenario-name
  DEPENDENCY artery-lte
  WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}/path-to-scenario)
  ```

- Following from the example of the already implemented scenario `run-simulte_cars` you should be able to determine the required items for defining your new scenario. This can be found under `path-to-project/OpenCV2X/scenarios/cars`
- Before attempting to make the scenario ensure that you remake your cmake environment as shown in the installation for artery tab.
- At this point you will be able to run the scenario by calling `cmake --build cmake-directory-name --target run_scenario-name`

2.3 Debugging Artery based simulations

2.3.1 Requirements:

- Clion IDE → https://www.jetbrains.com/clion/

2.3.2 Setting up the IDE:

- Install CLion following the below link will describe the installation do not use snap:

  https://www.jetbrains.com/help/clion/installation-guide.html

  Using snap means the environment variables won't be available and this makes things much more difficult.

- Pull the most recent version of OpenCV2X

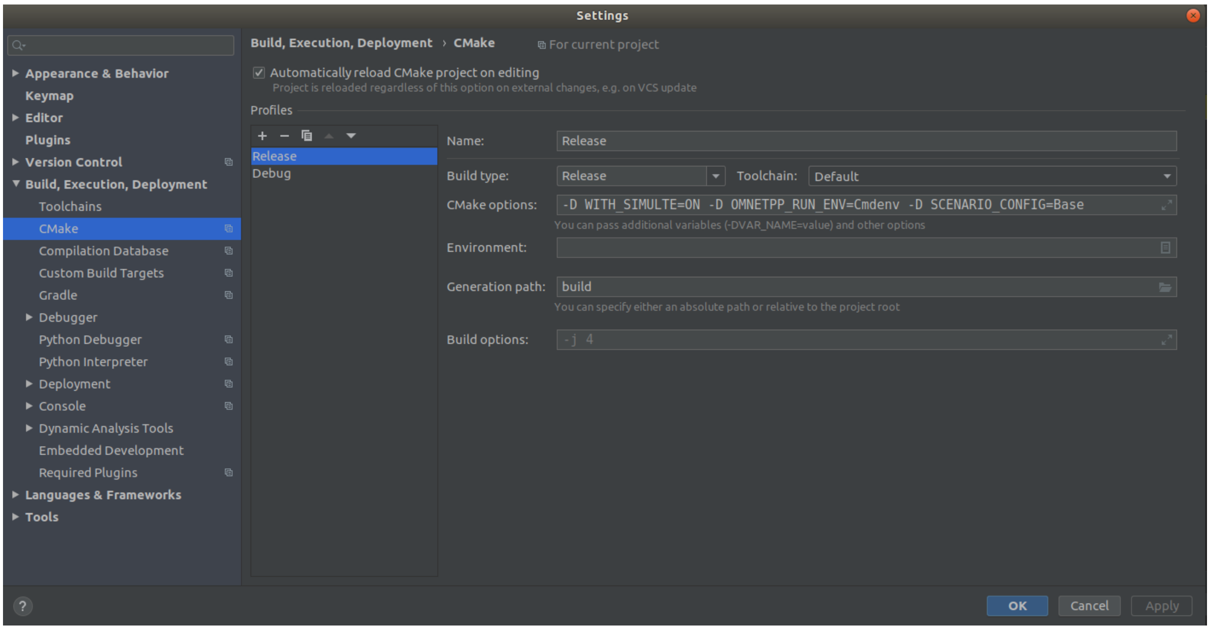
- Make all the individual projects from the root of the project:

  - `cd extern/inet & make WITH_OSGEARTH=no MODE=debug`
  - `cd extern/veins & ./configure & make`
  - `make simulte`
  - `make vanetza`

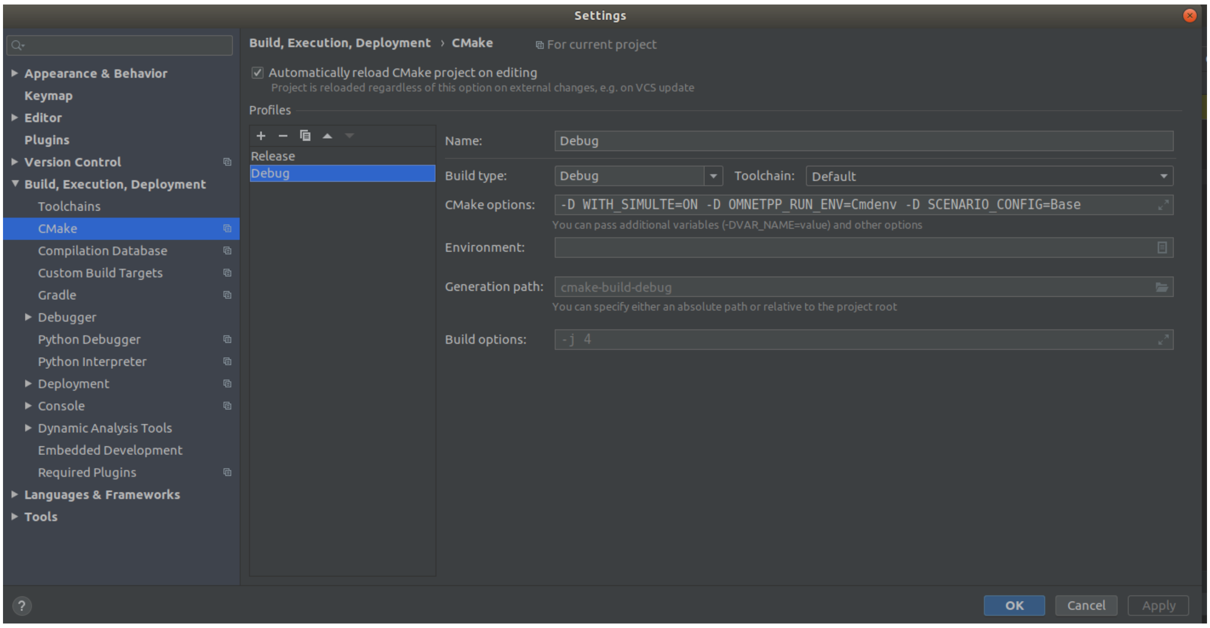
At this point the project should be ready to go now go to `File | Settings | Build, Execution, Deployment | CMake` here you will add a `Debug` and `Release` option

The configuration of each should look like the below images.

Release



Debug



2.3.3 Release configurations:

This is less concerning using the green hammer next to the top right configurations, this will automatically build it the correct way without need to define the configuration parts.

2.3.4 Debug configurations:

While this is complex it only needs to be done once and your system will be ready to run the debug runs and give full breakpoint functionality.

The below are what each field should show for the debug configuration of Mode 4:

Target: `debug_simulte-cars`

Executable: `opp_run_dbg`

This you will find in your OMNeT++ bin folder here is mine as an example: `/home/brian/omnetpp-5.5.1/bin/opp_run_dbg`

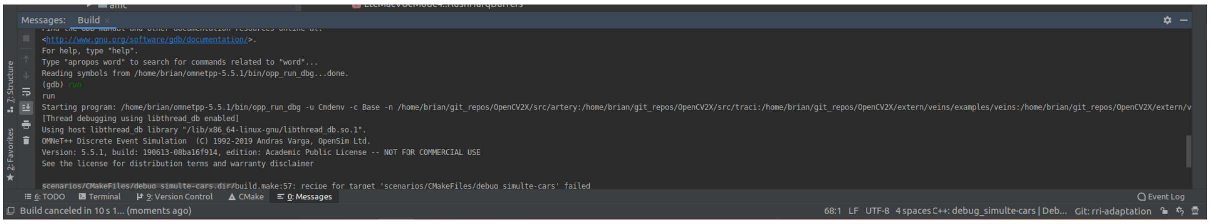
Below is the Program Arguments Section:

```
-u Cmdenv
-c Base
-n /home/brian/git_repos/OpenCV2X/src/artery:/home/brian/git_repos/OpenCV2X/src/traci:/home/brian/git_repos/OpenCV2X/extern/vei
-l /home/brian/git_repos/OpenCV2X/extern/inet/out/gcc-debug/src/libINET_dbg.so
-l /home/brian/git_repos/OpenCV2X/extern/simulte/out/gcc-debug/src/liblte_dbg.so
-l /home/brian/git_repos/OpenCV2X/cmake-build-debug/scenarios/highway-police/libartery_police.so
-l /home/brian/git_repos/OpenCV2X/cmake-build-debug/src/artery/envmod/libartery_envmod.so
-l /home/brian/git_repos/OpenCV2X/cmake-build-debug/src/artery/storyboard/libartery_storyboard.so
-l /home/brian/git_repos/OpenCV2X/cmake-build-debug/src/artery/libartery_core.so
-l /home/brian/git_repos/OpenCV2X/extern/veins/out/gcc-debug/src/libveins_dbg.so
 omnetpp.ini
```

Working Directory: `/home/brian/git_repos/OpenCV2X/scenarios/cars`

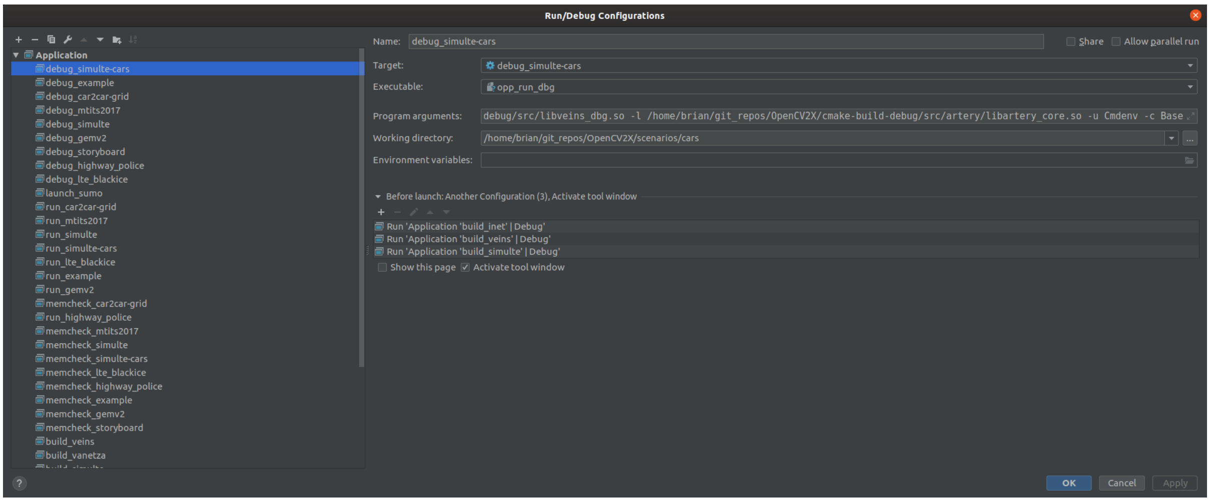
The above is the example for my version and will need to change but this is it in essence.

This can be generated easily by selecting the `debug_simulte-cars` configuration in the main window of CLion and clicking the `Build` button (the hammer just to the right of the configuration window). This will launch a **gdb** debugging session. When prompted simply enter run and then it will start the program with all the necessary information from above in it, see the image below `Starting program: /home....` After the opp_run_gdb section simply copying everything after this and pasting it into the Program Arguments Section should get you up and running.



Before launch: Another Configuration, Activate tool window: This can be left empty or can include a `build_inet`, `build_veins`, and `build+simulte` step this ensures you don't have to constantly be building the projects separately, except for `vanetza` which has to be dealt with separately if you wish to debug `vanetza` you must set it up correctly first, I will outline this below.

Ultimately the configuration window should look something like the below: the build before launch steps can be removed if they prove to cause issues.



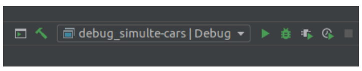
At this point in the top right corner you should be able to select `debug_simulte-cars | Debug` and then click the `bug` symbol to run in debug mode.

Under the file `OpenCV2X/extern/simulte/src/Makefile` change line `19` from `-lINET` to `-lINET_dbg`

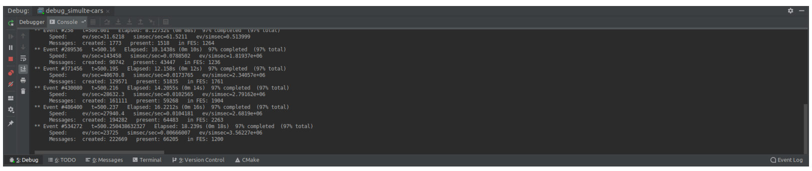
```
# Additional libraries (-L, -l options)
LIBS = $(LDFLAG_LIBPATH)$(INET_PROJ)/out/$(CONFIGNAME)/src  -lINET_dbg
```

At this point you should be able to run the debug configurations of the simulations.

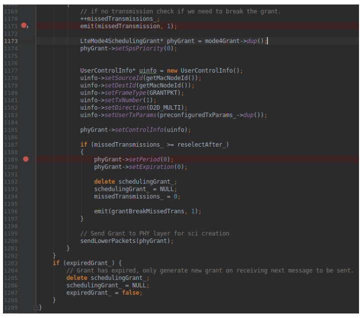
2.3.4 Launching debug simulations, adding breakpoints and debugging.



The above is what your launch window should look like with the specific options being set, we use the `debug` option of the `bug` next to the `play` button to launch or debug simulations.

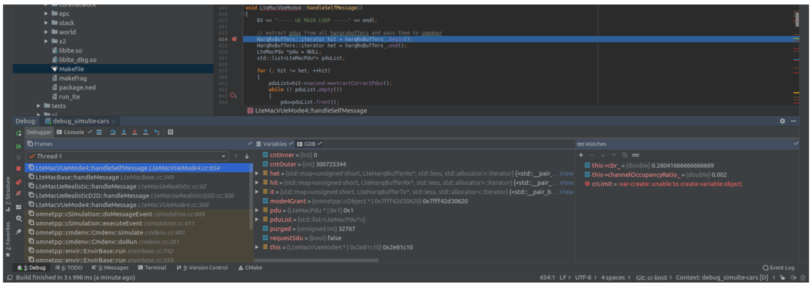


Above shows what happens when a debug simulation begins the `debug_simulte-cars` option should launch a window at the bottom of the screen, the console will show you how the simulation is running and the debugger window which is



Above shows some breakpoints in the code, when we want to add a breakpoint we simply left click next to the line numbers. A `red circle` in this case indicates a breakpoint.

shown below will show you things such as variables, function calls and other useful debug information.



The above as described previously is the debug window, in this case we have run into a breakpoint and can now debug the simulation. Most of this is self explanatory.

2.3.5 Debugging Vanetza

To debug vanetza you need to build it in debug mode and then launch your simulations, this cannot be done through CLion you have to do it manually which is a pain but should be doable.

If you look at the output of running `make vanetza` from the root directory, then you will see the following output in the first line

`cd extern/vanetza/build; cmake -DCMAKE_BUILD_TYPE=Release -DBUILD_SHARED_LIBS=ON ..` simply running the following instead of that will get you a debug version

`cd extern/vanetza/build; cmake -DCMAKE_BUILD_TYPE=Debug -DBUILD_SHARED_LIBS=ON ..` All we've done is swapped in the Debug as opposed to Release. This must be done every time you want to test changes to ensure that the correct version is used but allows full debugging capability.

3. Running and getting results:

Scenarios can be run in the way that is described in previous parts of this paper see Section 1.1.5/2.1.5.

Results generation is based on the documentation provided by OMNeT++ described in the following https://docs.omnetpp.org/tutorials/pandas/

It has the following requirements:

- Python3
- Numpy
- Pandas
- Matplotlib

Other technologies like R can be used but no documentation will be provided.

3.1 Parsing the Scalar and Vector files:

Results are parsed from the Scalar and vector files using the `scavetool` provided by OMNeT++ the below command shows how you can convert them into a csv file.

```
scavetool x *.sca *.vec -o test.csv
```

3.2 Importing parsed results

I would recommend as does the above tutorial using the Jupyter-notebook utility, this is provided most conveniently through Anaconda https://www.anaconda.com/

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

def parse_if_number(s):
    try: return float(s)
    except: return True if s=="true" else False if s=="false" else s if s else None

def parse_ndarray(s):
    return np.fromstring(s, sep=' ') if s else None

df = pd.read_csv('test.csv', converters = {
    'attrvalue': parse_if_number,
    'binedges': parse_ndarray,
    'binvalues': parse_ndarray,
    'vectime': parse_ndarray,
    'vecvalue': parse_ndarray})
```

The above is based on the earlier mentioned tutorial but it will load in the parsed file with both vectors and scalars included in the results of that file.

3.3 Calculating PDR

The below are the vectors of interest when trying to calculate PDR (we are looking at PDR over distance between the transmitter and receiver as our statistic)

```
pdr_vector = 'tbDecoded:vector'
pdr_dist_vector = 'txRxDistanceTB:vector'
```

The values which are recorded by `tbDecoded:vector` are the following `1 = successfully decoded`, `0 = failed to decode`, `-1 = no TB received` (this occurs when an SCI is sent without an accompanying TB). The same applies to other vectors which are recorded by the model e.g. `sciDecoded:vector` `1 = successfully decoded`, `0 = failed to decode` Other different parameters describe the cause of failures such as `tbFailedDueToNoSCI` this represents the fact that the TB was not decoded due to the SCI not being decoded, in this case `1 = Failed due to SCI not being decoded` and `0 = no failure by this cause`

The below will filter out all rows from the overall data frame which are not of interest to us and leave us only with the two variables we want in a single merged pandas DataFrame.

```
distances = df[(df["name"] == pdr_dist_vector) & (df["vectime"].notnull())]
decoded   = df[(df["name"] == pdr_vector) & (df["vectime"].notnull())]
distances = distances[["module", "vecvalue"]]
distances.rename(columns={"vecvalue": "distance"}, inplace=True)
decoded   = decoded[["module", "vecvalue"]]
decoded.rename(columns={"vecvalue": "decode"}, inplace=True)

new_df = pd.merge(distances, decoded, on='module', how='inner')
```

The code block below will then allow for the data to be parsed into PDR easily in 10 meter chunks up to 500 meters i.e. average PDR at 0-10, 10-20, 20-30 … 490-500m

```
bins = []
for i in range(50):
    bins.append({"count": 0, "success": 0})

for row in new_df.itertuples():
    for i in range(len(row.distance)):
        if row.distance[i] < 500:
            # Ensures that we have everything in 10m chunks
            remainder = int(row.distance[i] // 10)
            if row.decode[i] >= 0:
                # Only count TBs sent i.e. -1 will be ignored in result
                bins[remainder]["count"] += 1
                bins[remainder]["success"] += row.decode[i]
```

```
pdrs = []
distances = []
distance = 0
for dictionary in bins:
    pdrs.append((dictionary["success"] / dictionary["count"] * 100))
    distances.append(distance)
    distance += 10
```

The below will graph the results which have been parsed earlier using the matplotlib library for Python3.

```
fig, ax = plt.subplots()

ax.plot(distances, pdrs, label="PDR")

ax.set(xlabel='Distance (m)', ylabel="Packet Delivery Ratio %")
ax.legend(loc="lower left")
ax.tick_params(direction='in')

ax.set_xlim([0, (max(distances) + 1)])
ax.set_ylim([0, 101])
plt.xticks(np.arange(0, (max(distances))+50, step=50))
plt.yticks(np.arange(0, (101), step=10))

plt.show()
plt.savefig("test.png", dpi=300)
plt.close(fig)
```