

# Using Vissim External Driver Model DLL for Modeling Mixed Traffic with both Automated Vehicles and Human Driven Vehicles

Yunpeng Shi

Qing He

University of Buffalo, SUNY

## Table of Content

1. Introduction .....	3
2. Interface .....	3
3. Data .....	5
3.1 Data Overview .....	5
3.2 Create Variables .....	6
3.3 Extract and Replace Data .....	8
3.4 Signal Data .....	8
3.5 Data of Nearby Vehicles .....	8
3.6 Lane Change .....	9
4. Execute Command .....	9
5. Build Solution .....	10
6. Vissim Setup .....	10
6.1 Assign Vehicle Type .....	10
6.2 Assign Vehicle Input .....	11
6.3 Before Simulation .....	12
7. Run Sample .....	13
8. Appendix .....	15
9. Reference .....	19

## 1. Introduction

PTV Vissim is a traffic simulation software used for microscopic simulation. The default car-following model types installed in VISSIM are Wiedemann 74 and Wiedemann 99 which are popular in simulating human driving behaviors. If one wants to use other behavior models such as adaptive cruise control (ACC) or intelligent driver model (IDM) for connected and autonomous vehicles, a function called Driver Model DLL can be used to modified the vehicles' behaviors in simulation. The External Driver Model allows users to replace various vehicle-related information such as velocity, acceleration, location, lane-changing signal, and intersection signals for vehicles in the simulation runs.

This tutorial will explain how to write into the External Driver Model DLL interface and use it in Vissim. Towards the end, a sample code is provided. Additionally, a Vissim sample model is attached for running the sample code. This tutorial is constructed based on Microsoft Windows 10 pro, PTV Vissim 7.00-13, and Microsoft Visual Studio Community 2017 Version 15.4.0.

## 2. Interface

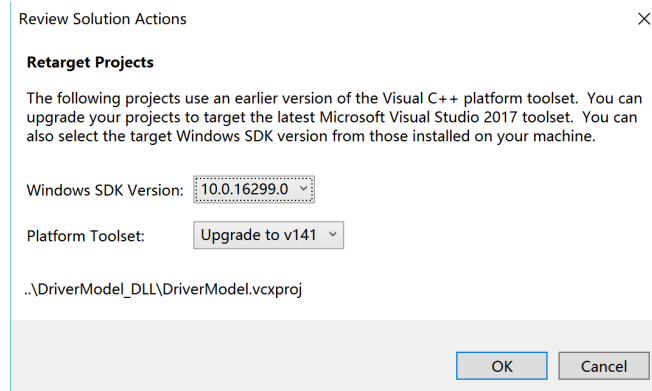
This section introduces the basics of the driver model DLL interface.

Generally, the External Driver Model DLL interface file can be found in the following folder: Local Disk (C) / Program File/ PTV Vision/ PTV Vissim/ API/ DriverModel\_DLL. There are several files in the folder:

- DriverModel.h: Header file for a driver model DLL.
- DriverModel.cpp: Main source file of a driver model DLL.
- DriverModel.vcxproj: Visual C++ 2010 project file for a driver model DLL. This file can be used if the DLL is to be created with Microsoft Visual C++.
- Interface Description: An explanatory PDF introducing the content in driver model DLL.
- DriverModel.Changes: A text file explaining the changes in driver model DLL

Some contents from this tutorial are taken from the Interface Description. Visual C++ is required to use driver model DLL.

Open DirverModel.vcxproj with Visual C++, a Review Solution Action message might show. The settings in the figure below works.



After opening the project file, find “Solution Explorer” (usually on the left of the window), click on **DriverModel**, and then click on **DriverModel.cpp**, the default DLL code will appear. All codes will run once per vehicle per time step, meaning that the calculated values in DLL will be update for each vehicle in each time step in Vissim. There are three main functions: `DriverModelSetValue`, `DriverModelGetValue`, `DriverModelExecuteCommand`. The format and meaning of the functions are as follows:

```
DRIVERMODEL_API int DriverModelSetValue (long    type,
                                         long    index1,
                                         long    index2,
                                         long    long_value,
                                         double  double_value,
                                         char    *string_value)
{
    switch (type) {
        All the cases
    }
}
```

The function for Set Value is to extract data from Vissim. As defined in the Interface Description, VISSIM passes the current value of the data item indicated by `type` and (for most types) indexed by `index1` and sometimes `index2`, the meaning of these values will be explained later in the tutorial. The value is passed in `long_value`, `double_value` or `string_value`, depending on `type`. Under the switch function are all the cases or types of data you can get from Vissim to simulate the vehicle behavior. The specification of the cases will be explained later in the tutorial.

```
DRIVERMODEL_API int DriverModelGetValue (long    type,
                                         long    index1,
                                         long    index2,
                                         long    *long_value,
                                         double  *double_value,
                                         char    **string_value)
{
```

```

switch (type) {
All the cases
}
}

```

The Get Value function has the same format as Set Value, but can send data back to Vissim.

```

DRIVERMODEL_API int DriverModelExecuteCommand (long number)
{
switch (number) {
case DRIVER_COMMAND_INIT :
return 1;
case DRIVER_COMMAND_CREATE_DRIVER :
return 1;
case DRIVER_COMMAND_KILL_DRIVER :
return 1;
case DRIVER_COMMAND_MOVE_DRIVER :
return 1;
}
}

```

The Execute Command is the core function in DLL for calculating vehicle behavior. The commands mean as follows:

- **Init** is called at the start of a VISSIM simulation run to initialize the driver model DLL.
- **Create driver** is called from VISSIM during the simulation run whenever a new vehicle is set into the network in VISSIM at the start of each time step.
- **Kill driver** is called from VISSIM when a vehicle reaches its destination and thus leaves the network to free memories.
- **Move driver** is called from VISSIM during the simulation run once per time step for each vehicle of a vehicle type which uses this driver model DLL. All vehicle behavior calculations should be written under this command.

All of the four commands have to return 1 as shown on the top example for Vissim to stop the simulation run. Create driver and kill driver commands can usually be controlled by Vissim if no special requests are needed.

The order of the driver model DLL commands being called at each time step is: **Set Value** to extract data needed from Vissim, **Execute Command (Move Driver)** to calculate driver behavior inserted in the command, then **Get Value** to send calculated parameters back to Vissim.

### 3. Data

This section introduces the data that can be extracted from and sent to Vissim.

### 3.1 Data Overview

As explained in the previous section, Set Value and Get Value are used to exchange parameter data with Vissim, the data includes:

- Subject vehicle data: physical characteristics of the current vehicle such as velocity, acceleration, location, and size; decision parameters such as turning indicator and desired velocity
- Nearby vehicle data: Information about nearby vehicles such as ID, velocity, acceleration, position, distance from the current vehicle. The detection range is two lanes to both sides, and two vehicles upstream and downstream.
- Link information data: Lane width and ending distance from current location.
- Current and upcoming environment data: Information about current slopes or upcoming curve.
- Next signal head data: Information about distance and state of the upcoming signal head.
- Speed decision data: Parameters related to speed limit sign.
- Behavior suggestion data: The behavior parameters that will influence decisions in the next time step, such as desired acceleration and lane change.

There are two ways to see the full list and explanation of the data.

Option 1: Interface Description. PDF, page 7-12.

Option 2: In **DriverModel.vcproj**, go to the Solution Explorer on the left, open **DriverModel** → **DriverModel.h**. The window will look like the figure below, all the parameters are explained in green words below the purple parameter names.

```
25 #endif
26
27 /*=====*/
28
29 /* general data: */
30 /* (index1, index2 irrelevant) */
31 #define DRIVER_DATA_PATH 101
32 /* string: absolute path to the model's data files directory */
33 #define DRIVER_DATA_TIMESTEP 102
34 /* double: simulation time step length [s] */
35 #define DRIVER_DATA_TIME 103
36 /* double: current simulation time [s] */
37 #define DRIVER_DATA_PARAMETERFILE 104
38 /* string: name (including absolute path) of a vehicle type's parameter file */
39 #define DRIVER_DATA_STATUS 105
40 /* long:
41 /* 0=OK,
42 /* 1=Info (there's some further information available),
43 /* 2=Warning (there are warnings available),
44 /* 3=Error (an recoverable error occurred but simulation can go on),
45 /* 4=Heavy (an unrecoverable error occurred and simulation must stop)
46 /* (used by DriverModelGetValue())
47 #define DRIVER_DATA_STATUS_DETAILS 106
48 /* string: info/warning/error message for nonzero status
49 /* (used by DriverModelGetValue())
50 /* (is retrieved by VISSIM only if DRIVER_DATA_STATUS is nonzero)
51
52 /* current vehicle driver unit data (VDU to be moved next): */
53 /* (index1, index2 irrelevant) */
54 #define DRIVER_DATA_VEH_ID 201
55 /* long: vehicle number */
56 #define DRIVER_DATA_VEH_LANE 202
57 /* long: current lane number (rightmost = 1) */
58 #define DRIVER_DATA_VEH_ODOMETER 203
59 /* double: total elapsed distance in the network [m] */
60 #define DRIVER_DATA_VEH_LANE_ANGLE 204
61 /* double: angle relative to the middle of the lane [rad] */
```

### 3.2 Create Variables

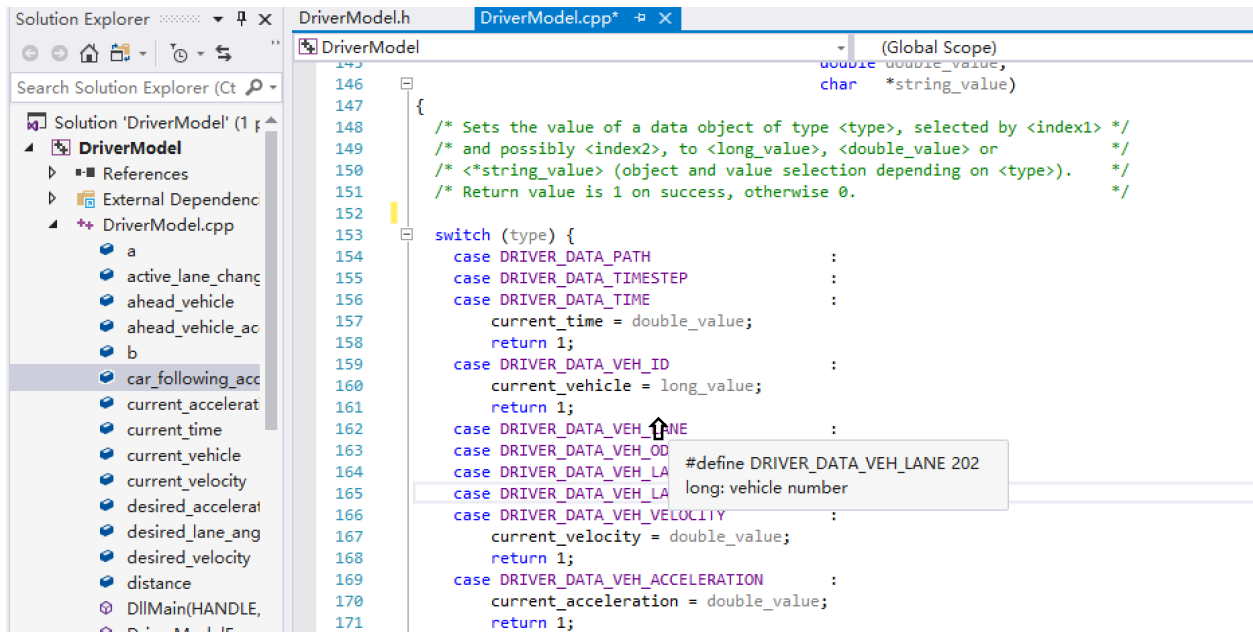
When data information is read from Vissim by Set Value, variables can be created to save the information. In order to do that, go to **DriverModel.cpp**, create variables in the blank space prior to Set Value function, make sure that these codes are not in any of the three main functions. The format should be (data format) + space + variable name you create. An example is shown in the figure below.

```
--
45  /*Create Variables*/
46
47  double current_time;
48  long current_vehicle;
49  double current_velocity;
50  double current_acceleration;
51  double vehicle_length;
52  double vehicle_x_coord;
53  double vehicle_y_coord;
54  long vehicle_type;
55  int vehicle_current_link;
--
```

Parameters in Vissim have default formats, format of the variables should match the default format in order to avoid errors. The default formats are shown in the parameter explanation in **DriverModel.h**. Figure below is an example, **DRIVER\_DATA\_TIME** has explanation: **double: current simulation time [s]**, indicating that parameter “current simulation time” should be saved in a variable with format double. Thus, the variable can be created as: **double current\_sim\_time**.

```
31  #define DRIVER_DATA_PATH 101
32  /* string: absolute path to the model's data files directory */
33  #define DRIVER_DATA_Timestep 102
34  /* double: simulation time step length [s] */
35  #define DRIVER_DATA_TIME 103
36  /* double: current simulation time [s] */
```

In addition, there is a faster way in **DriverModel.cpp** to check the default data format. In Set Value function, move the cursor to the parameter below the desired one, a message box will appear, the second line in the box indicates the format and explanation of the desired parameter. For example, as shown in the figure below, the cursor is on **Driver\_DATA\_VEH\_LANE**, but the message box appeared shows explanation describing the previous parameter: **DRIVER\_DATA\_VEH\_ID**. The second line in the box shows that the format to save vehicle number should be long. Thus, variable saving vehicle number (“current\_vehicle” in this example) can be created by: **long current\_vehicle**.



### 3.3 Extract and Replace Data

After creating variables to save data, use Set Value function to exact data from Vissim. The usual steps are:

1. Find the desired parameter (**case**) in Set Value function, press Enter to create a blank line.
2. Type in: “created variable = format\_value;”. The format should match the default format mentioned previously.
3. In the next line, type in “return 1;”.

An example is shown below extracting current vehicle acceleration from Vissim in Set Value function and saving to the variable current\_acceleration.

```
case DRIVER_DATA_VEH_ACCELERATION:
    current_acceleration = double_value;
    return 1;
```

After running Execute Commands, calculated parameters can be sent back to Vissim using Get Value function. The steps are similar to the Set Value function:

1. Find the desired parameter (case) in Get Value function, if the desired parameter is not found, copy the case from Set Value function, then press Enter to create a blank line.
2. Type in: “\*format\_value = created variable;”. The format should match the default format mentioned previously.
3. In the next line, type in “return 1;”.

An example is shown below using the same variable as the previous example.

```
case DRIVER_DATA_VEH_ACCELERATION:
    *double_value = current_acceleration;
```



```
return 1;
```

### 3.4 Signal Data

The data describing intersection signal state is discrete. If the intersection is signalized, the value means as follows:

Set DRIVER\_DATA\_SIGNAL\_STATE =  
red = 1, amber = 2, green = 3, red/amber = 4, amber flashing = 5, off = 6,  
other = 0

If the intersection is yield sign, the value means as follows:

Set DRIVER\_DATA\_SIGNAL\_STATE =  
red = 1, green = 3

### 3.5 Data of Nearby Vehicles

Data of nearby vehicles can be obtained in Set Value function, including vehicle information, turning indicator, category information, and relative position to current vehicle. In Set Value, nearby vehicle data starts with DRIVER\_DATA\_NVEH. The range is up to two vehicles each downstream and upstream, on up to 2 lanes on both sides and the current lane. Index 1 and index 2 mentioned in section 2 are used to specify the chosen vehicle as follows:

For DRIVER\_DATA\_NVEH\_\* index1 and index2 are used as follows:  
index1 = relative lane: +2 = second lane to the left, +1 = next lane to the left,  
0 = current lane,  
-1 = next lane to the right, -2 = second lane to the right  
index2 = relative position: positive = downstream (+1 next, +2 second next)  
negative = upstream (-1 next, -2 second next)

For example, if one wants to get the distance between current vehicle and the front vehicle on the next lane to the left, the code can be as follows:

```
case DRIVER_DATA_NVEH_DISTANCE :  
    if (index1 == 1 && index2 == 1)  
    {  
        gross_distance = double_value;  
    }  
    return 1;
```

### 3.6 Lane Change

The external driver model DLL allows current vehicle to perform lane change in the simulation. There are two ways to achieve that:

1. Simple lane change: Vissim will have control over the lane change process, a lane changing signal needs to be sent to Vissim by driver model DLL. In Get Value function, find simple\_lanechange case and make it as the following format,

```
case DRIVER_DATA_SIMPLE_LANECHANGE :  
    *long_value = 1;  
    return 1;
```

Then find DRIVER\_DATA\_ACTIVE\_LANE\_CHANGE, set the value to 1 to change to the left or -1 to change to the right. Vissim will make the lane change when default safety conditions are met.

2. Full control lane change: The driver model DLL will have full control in the lane changing process. For details, please look at Interface Description PDF Page 15.

#### 4. Execute Command

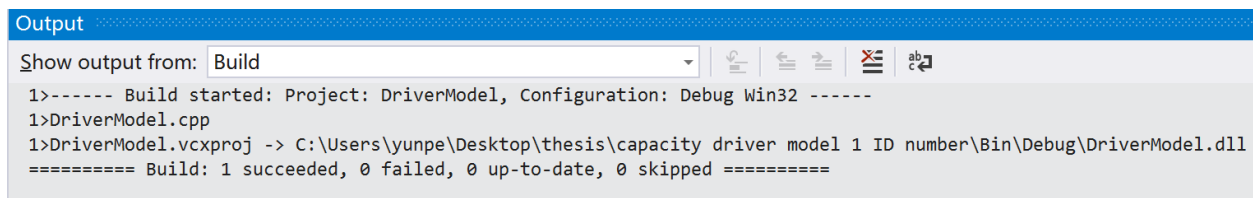
This section introduces Execute Command.

As introduced in Section 2, calculation codes should be included in Execute Command. Other than create or delete drivers (vehicles), codes should be under `case DRIVER_COMMAND_MOVE_DRIVER`. A complete process of creating an IDM driver model using Driver Model DLL is in Appendix.

#### 5. Build Solution

This section explains how to build the DLL file.

After finish building the model in DriverModel.cpp, select top task bar **Build → Build Solution** (or press F7). The output window on the bottom will show the build outcome. If failed, please debug the driver model. If succeed, a path showing where the DriverMode.dll file is stored will appear like the figure below. Save the path for future use (inputting external driver model into Vissim).



```
Output  
Show output from: Build  
1>----- Build started: Project: DriverModel, Configuration: Debug Win32 -----  
1>DriverModel.cpp  
1>DriverModel.vcxproj -> C:\Users\yunpe\Desktop\thesis\capacity driver model 1 ID number\Bin\Debug\DriverModel.dll  
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
```

## 6. Vissim Setup

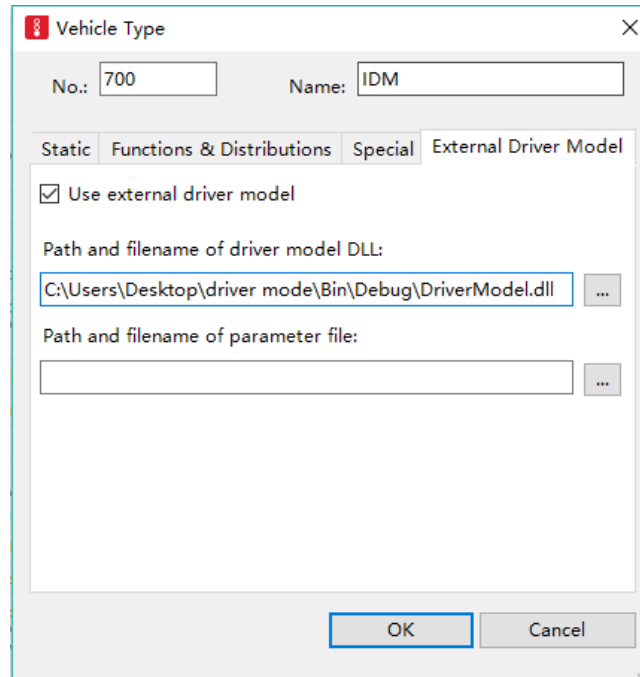
The section will explain how to insert the Driver Model DLL into Vissim.

### 6.1 Assign Vehicle Type

After building the Driver Model DLL, open Vissim. Click on the top task bar **Base Data** then **Vehicle Types**. The driver model DLL can either be applied on current vehicle types or new vehicle types created. To create a new vehicle type, right click on blank space inside the Vehicle Types window, and click on **Add...**

A vehicle type setting will appear after creating a new vehicle type or if use a current vehicle type, right click on the designated vehicle type and click **Edit...**

In vehicle type settings, click on **External Driver Model** tab, check the box “**Use external driver model**”. Then type in or browse the path of the DriverModel.dll file. Then click **OK**. An example is shown below.



### 6.2 Assign Vehicle Input

After entering the vehicle type, click on the top task bar **Traffic** then **Vehicle Composition**. The Figure below is an example of vehicle composition window. The left window contains different vehicle compositions. Use an existing one by left click it or create a new one by right click in the blank space and then choose **Add...** Then move to the window on the right which shows relative flows between different vehicle types. If only one type of vehicle is used, click on the second column **VehType** and choose the desired vehicle type. If more than one type is used, right click in the blank space and choose **Add...** to create as many types as need, then choose desired

vehicle types in the second column **VehType**. The relative flow can be modified in the fourth column **RelFlow**, inserting either the percentage of each vehicle type or the desired volume of each vehicle type will be acceptable. Vissim will automatically transform this relative flow to percentages. The desired speed distribution can be edited in the third column, to create new desired speed distributions, go to top task bar **Base Data** → **Distributions** → **Desired Speed**.

Cou	No	Name
1	1	Default
2	2	Autonomous
3	3	human
4	4	NBAUTONOMOUS
5	5	Line1
6	6	Line2

Count:	VehType	DesSpeedDistr	RelFlow
1	100: Car	1048: car 50km	270.000
2	700: IDM	1049: auto 50k	2430.000

After creating the desired vehicle compositions, click on the top task bar **Lists** → **Private Transport** → **Input** to enter the desired volume. Choose the desired link on the fourth column, type in desired volume in the fifth, and choose the vehicle composition just created in the last column.

Count:	No	Name	Link	Volume(0)	VehComp(0)
1	1	link 1	1	0.0	5: Line1
2	2	link 2	2	3000.0	5: Line1
3	4	2human	2	0.0	3: human

### 6.3 Before Simulation

Before simulation, go to top task bar **Simulation** > **Parameters**. As shown in the figure below, if any vehicle in the simulation use driver model DLL, the last option **Number of cores** needs to be 1 Core to avoid errors in the simulation. After done all the steps above, simulation can be started.

Simulation Parameters

Comment:

Period:  Simulation seconds

Start Time:  [hh:mm:ss]

Start Date:  [DD.MM.YYYY]

Simulation resolution:  Time step(s) / Sim. sec.

Random Seed:

Number of runs:

Random seed increment:

Dynamic assignment volume increment:  %

Simulation speed: ☐ 5.0 Sim. sec. / s  
☒ maximum  
☐ Retrospective synchronization

Break at:  Simulation seconds

Number of cores:

OK Cancel

## 7. Run Sample

This section will guide you to run the Vissim sample.

The sample contains two parts, sample codes and sample Vissim model.

The sample dll model is an IDM model similar to the one in Appendix. To make sure that the dll file works, it needs to be rebuilt. Open **DriverModel.vcxproj** in **Sample Code** file, select top task bar **Build → Build Solution** (or press F7). The output window will show a succeed build with a path of the dll file. Copy the path.

Next, open **Capacity Analysis** (Vissim Input Model) in **Sample Vissim Model** file. Select top task bar **Base Data → Vehicle Types**. Right click on No 700 IDM, and select **Edit...** A vehicle type window will appear. Choose **External Driver Model**, and update the box below “**Path and filename of driver model DLL:**” with the path you just copied.

Then choose top task bar **Simulation → Continuous** to start the simulation. This sample Vissim model is a two-lane throughway network with length of 1,000 meters. Without changing any other parameters, the simulation will run for one hour. Both lanes will have desired volume

of 2000 vehicle/hour/lane, and vehicle composition of 50% Vissim default human driven vehicles (Wiedemann 99), and 50% vehicles with IDM behavior.

\* Author used Vissim 7 and Visual Studio (C++) 2017. Other versions might require minor changes to make the sample work.

## 8. Appendix

### Intelligent Driver Model (IDM)

The IDM acceleration calculation is as follows:

$$a_{IDM}(S, v, \Delta v) = a \left[ 1 - \left( \frac{v}{v_0} \right)^\delta - \left( \frac{S^*(v, \Delta v)}{S} \right)^2 \right]$$

And

$$S^*(v, \Delta v) = S_0 + vT + \frac{v\Delta v}{2\sqrt{ab}}$$

$S$ : bumper to bumper distance to the leading vehicle

$v$ : actual speed

$\Delta v$ : velocity difference from the leading vehicle.

The values of the three parameters above will use values in Vissim simulations, other parameter values are fixed in this example, use values shown as follows:

Maximum Acceleration $a$	$2 \text{ m/s}^2$
Maximum Deceleration $b$	$3 \text{ m/s}^2$
Desired Speed $v_0$	15 m/s
Free Acceleration Component $\delta$	4
Jam Distance $S_0$	1.5 m
Safe-time Headway $T$	0.6 s

Codes:

### Default Parameters (by Vissim)

```
/*Default Parameters*/  
  
double desired_acceleration = 3.5;  
double desired_lane_angle   = 0.0;  
long    active_lane_change   = 0;  
long    rel_target_lane      = 0;  
double  desired_velocity     = 13.9;  
long    turning_indicator    = 0;  
long    vehicle_color        = RGB(225,225,225);
```

### Create Variables

```
/*=====*/  
/*Create Variables to store data*/
```

```

long current_vehicle;
double current_velocity;
double vehicle_length;
double gross_distance;
double speed_difference;
double s_star;
double car_following_acceleration;
double ahead_vehicle;
static int vehicle_to_stop = 0;

```

## IDM Parameters and Equations

```

/*=====*/
/* IDM parameters and equation*/

double a = 2;           // maximum acceleration, constant a (m/s^2)
double b = 3;           // maximum deceleration, constant b (m/s^2)
double IDM_desired_velocity = 15; // desired velocity used in IDM (m/s)
double v0 = IDM_desired_velocity;
double jam_distance = 0.6; // minimum gross distance (m)
double S0 = jam_distance;
double T = 0.6;           // safe-time headway (s)

double S_star(double v, double v_delta)
{
    double S_star = S0 + (v * T) + ((v * v_delta) / (float)(2 * sqrt(a*b)));
    return S_star;
}

```

## Set Value (Unused cases are deleted to avoid confusion)

```

DRIVERMODEL_API int DriverModelSetValue (long type,
                                          long index1,
                                          long index2,
                                          long long_value,
                                          double double_value,
                                          char *string_value)
{
    switch (type) {
        case DRIVER_DATA_VEH_ID :
            current_vehicle = long_value;
            return 1;
        case DRIVER_DATA_VEH_VELOCITY :
            current_velocity = double_value;
            return 1;
        case DRIVER_DATA_VEH_ACCELERATION :
            current_acceleration = double_value;
            return 1;
        case DRIVER_DATA_VEH_LENGTH :
            vehicle_length = double_value;
            return 1;
        case DRIVER_DATA_NVEH_ID :

```



```

        if (index1 == 0 && index2 == 1) {
            ahead_vehicle = long_value;
        }
        return 1;
    case DRIVER_DATA_NVEH_DISTANCE :
        if (index1 == 0 && index2 == 1)
        {
            gross_distance = double_value;
        }
        return 1;
    case DRIVER_DATA_NVEH_REL_VELOCITY :
        if (index1 == 0 && index2 == 1)
        {
            speed_difference = double_value;
        }
        return 1;
    case DRIVER_DATA_DESIRED_ACCELERATION :
        desired_acceleration = double_value;
        return 1;
    case DRIVER_DATA_DESIRED_LANE_ANGLE :
        desired_lane_angle = double_value;
        return 1;
    case DRIVER_DATA_ACTIVE_LANE_CHANGE :
        active_lane_change = long_value;
        return 1;
    case DRIVER_DATA_REL_TARGET_LANE :
        rel_target_lane = long_value;
        return 1;
    default :
        return 0;
}
}

```

## Get Value (Unused cases are deleted to avoid confusion)

```

DRIVERMODEL_API int DriverModelGetValue (long type,
                                          long index1,
                                          long index2,
                                          long *long_value,
                                          double *double_value,
                                          char **string_value)
{
    switch (type) {
        case DRIVER_DATA_DESIRED_ACCELERATION :
            *double_value = desired_acceleration;
        default :
            return 0;
    }
}

```

## Execute Command

```

DRIVERMODEL_API int DriverModelExecuteCommand (long number)
{

    switch (number) {
        case DRIVER_COMMAND_INIT :
            return 1;
        case DRIVER_COMMAND_CREATE_DRIVER :
            return 1;
        case DRIVER_COMMAND_KILL_DRIVER :
            return 1;
        case DRIVER_COMMAND_MOVE_DRIVER :

            if (ahead_vehicle < 0) {
                vehicle_to_stop = current_vehicle;
            }

//In Vissim, if a vehicle passes the destination and is deleted from the network, the
vehicle number will show as -1.
//Since IDM is a car-following model, it cannot control the first vehicle of a stream
properly. Therefore, the control of the first vehicle in stream will be handled by
Vissim.

            if(current_vehicle != vehicle_to_stop ){

                double S = gross_distance - vehicle_length;
                double v = current_velocity;
                double v_delta = speed_difference;
                double s_star = S_star(current_velocity, speed_difference);

                car_following_acceleration = a * (1 - pow((v / (float)(v0)), 4) -
pow((s_star / (float)S), 2));

                if (gross_distance > jam_distance){
                    desired_acceleration = car_following_acceleration;
// setting the IDM acceleration as the desired acceleration if gross distance is greater
than the minimum gross distance required.
                }

            }
            return 1;
        default :
            return 0;
    }
}

```

## **9. Reference**

Interface Description.pdf, PTV Vissim