Data sampling and data storage for research simulators

CONTENTS

Introduction

Measuring driver performance data is one of the most important functions of a research driving simulator. A unique feature of the driving simulator software of Carnetsoft is that it allows the user to easily sample time-based safety margins that are considered to be very hard or even impossible to measure on the road, and are often impossible to measure in other driving simulators. The most important reason for this is that our software uses a high quality logical road network representation (the logical road database). Among these variables is the TLC (Time-to-line crossing) that is measured accurately both for straight road segments and for road curves.

There are three different types of data that can be measured and stored in this simulator:

1. **Raw data**, with a fixed sample frequency, typically 10 Hz. This is the typical type of data that is stored on most simulators. It is used for offline data analysis and inspection and consists of variables such as vehicle speed, yaw rate, time to line crossing, time headway etc.
2. **Processed data** that is measured and analyzed during the simulation and the results are written to an ascii file. This consists of data measured and stored in scripts and is typically of a higher level than data type 1, the raw data. Process data may consist of a standard deviation of lateral position on a certain stretch of road, or the brake reaction time in a specific event.
3. **Driving errors** as stored in the student assessment system that comes with the driver training simulations. This contains information of the highest level, such as the number of times the speed limit was exceeded, or time headway was too small.

This gives the researcher a wide range of data to be analyzed.

1. Raw data

Traffic.exe and the scenario scripting language generate functionality for binary datasampling and storing of variables in binary datafiles. The files can be read and processed by the Dataproc data visualization and analysis program. The scriptfile 'DataProc.sci' contains some predefined functionality for datasampling. DataProc.sci can be included in any script in which binary data sampling and –storage is needed. Although the data to be sampled and stored can be specified in script, it may be easier to use the application ExpPrep.exe for that purpose. See ExperimentSpecification.pdf for more details.

Scripts allows the user to:
- define the data to be sampled together with the sampling frequency
- create (open) and close a binary datafile
- set event-codes for inclusion in eventfiles.

Data are stored in a \data folder located in the same folder as where the script is located. So suppose there is a folder \myexperiment that contains the *.scb (binary scenario script) files, there MUST be a folder \data within \myexperiment. If there is no such folder, the data cannot be stored to file.

Here, defining and storing binary datafiles is explained.

1.1 Define the data

The datarecords that are sampled are defined with the procedures SetSampleFrequency, ClearDataVariables, AddDataVariable and AddDataFunction.

Here's a piece of script (from DataProc.sci) that uses these functions:

```
Proc( ClearDataVariables );
Proc( SetSampleFrequency, 10 ); // sample with 10 Hz to binary file
EXT_DataDef := DataDef();          // get the data specification filename
Proc( OpenFileRead, EXT_DataDef );
DataType := ReadNumber( EXT_DataDef );
While ( DataType >= 0 ) {
    If ( DataType = 1 or DataType = 2 ) {
        DataCode := ReadNumber( EXT_DataDef );
        Proc( AddDataVariable, DataCode );
     }
     Else {
        FuncName := ReadString( EXT_DataDef );
        Proc( AddDataFunction, FuncName );
      }
      DataType := ReadNumber( EXT_DataDef );
}
Proc( CloseFile, EXP_DataDef );
```

The function DataDef() returns the name of the data definition file (extension .dd) that was created with the ExpPrep.exe tool. In this example, the data definitions are read and added to te data storage system.

1.2 SetSampleFrequency

Proc(SetSampleFrequency, *number*);

For example, Proc(SetSampleFrequency, 10); This sets the sample frequency to 10 Hz (i.e. 10 samples are stored per second). It is advised to not set this value too high, since higher values result in larger binary datafiles. For normal purposes, 10 Hz is advised. By setting a sample frequency, all data are stored with the same sample interval. For example, by Proc(SetSampleFrequency, 10) all sample intervals are quaranteed to be 100 ms. To guarantee this, all values are measured in real time and extrapolated or interpolated such that a value is stored for each defined variable every 100 ms. Since analog variables are assumed, this extrapolation or interpolation is valid. However, if the user wants to sample discrete values that give for example the value 50 on one sample and 100 on the next sample, this method is less desirable. In cases like these one doesn't want interpolation of values. In cases like these it is better to store the values in an eventfile. As an alternative, sample frequency may be set to 0 by Proc(SetSampleFrequency, 0); Setting the sample frequency to 0 ensures that all data are stored in their raw format without extrapolation or interpolation with a sample frequency of 10 Hz.

1.3 ClearDataVariables

Proc(ClearDataVariables);

This procedure clears all data variables. Suppose you want to sample some variables while driving a specific script, but sample another set of variables while driving another script. You need to clear the data variables first and in-between.

1.4 AddDataVariable

Proc(AddDataVariable, *VarType*);

For example:

```
Proc( SetSampleFrequency, 10 );
Proc( ClearDataVariables );
Proc( AddDataVariable, d_velocity );
Proc( AddDataVariable, d_latpos );
Proc( AddDataVariable, d_gear );
```

In this example, the data definition script defines a record with 3 values (actually four, because the timestamp d_time is also stored automatically for each record): the vehicle velocity in m/s, the lateral position in meters and the gear position, in that order.

AddDataVariable adds a specific variable to the list of variables that will be sampled. A description of these variables is in the scriptfile 'DataProc.sci':

Symbolic constant	Id	Description
d_interval	2	interval between timeframes in StTraffic simulation: in ms.

d_velocity	3	velocity of MainTarget: in m/s
d_acc	4	acceleration of MainTarget: in m/s^2
d_latvel	5	lateral velocity of MainTarget: in m/s. If vehicle moves to LEFT: positive values. If vehicle moves to RIGHT: negative values
d_dtoint	6	distance to next intersection: in meters. Distance is measured from the center of the front bumper to the point where the intersectionplane starts.
d_dfrint	7	distance from last intersection: in meters. Distance is measured from the center of the front bumper.
d_dtoseg	8	distance to the end of the present segment: in meters. Distance is measured from the center of the front bumper.
d_latpos	9	Lateral distance between center of front bumber of MainTarget and the centerline of the Rightmost DLANE (DLane[0]). If the center of the front bumper is to the LEFT of this line: positive values. To the RIGHT of this line: negative values.
d_segnum	10	Segment number of MainTarget
d_pathnum	11	Path number of MainTarget
d_internum	12	Intersection number if MainTarget is on an intersection
d_scennum	13	The number of the first active scenario. Since there are usually a large number of scenarios active simultaneously, this number is of limited value.
d_gear	14	Gear position of MainTarget (0=free, 1..5)
d_indicator	15	Indicator status of MainTarget: 0: IndicatorOff 1: IndicatorRight 2: IndicatorLeft, 3: IndicatorAlarm
d_rpm	16	Engine RPM of MainTarget
d_steer_raw	17	The raw steering value from the cabin
d_gas_raw	18	The raw accelerator pedal value from the cabin
d_brake_raw	19	The raw brake pedal value from the cabin
d_clutch_raw	20	The raw clutch pedal value from the cabin
d_steer	21	Steering wheel angle (in degrees)
d_wheelangle	22	Front wheel angle (in degrees)
d_gas	23	Accelerator pedal position as a percentage (0..100)
d_brake	24	Brake pedal position as a percentage (0..100)
d_clutch	25	Clutch pedal position as a percentage (0..100)
d_steertorque	26	Steertorque in Nm (if supported by the cabinIO)
d_brakeforce	27	Brakeforce in N (if supported by the cabinIO)
d_xpos	28	X coordinate of center of frontbumper
d_ypos	29	Y coordinate of center of frontbumper

d_heading	30	Heading of vehicle with respect to the road (in degrees)
d_traflight	31	Status of trafficlight that MainTarget is approaching: 0: Absent 1: Green 2: Yellow 3: Red 4: YellowFlash 5: YellowRed (German intermediate phase) 6: Blank
d_time	32	current timeframe (timeframe is always stored automatically so does not have to be specified)
d_SpeedDif	33	The difference (in m/s) between the actual velocity and the maximum allowed velocity according to a set of rules. These rules are the rules used by the robot cars. So this is the speed difference from the normative model. If this value is positive, the MainTarget is driving too fast with respect to the normative model. Value is always >= 0.
d_LaneDirection	34	Indicates whether the lateral position is right or wrong with respect to the normative model. Values may be : 2: OK, or Straight 1: Right, MainTarget should move to right 3: Left, MainTarget should move to left
d_LowestSpeedCause	35	While driving, the actual speed may violate a number of different rules that prescribe a maxmum allowed speed. The rule that is most seriously violated (i.e. the difference between the actual velocity and the maximum velocity according to that rule is largest), is returned by d_LowestSpeedCause. -1 : No speed rule violated 0: S_ChangeLead: while changing lanes the speed is too high with respect to a lead vehicle on the new lane 1: S_Follow : speed too high with respect to lead vehicle 2: S_MaxVelocity : speed higher than the maximum allowed speed (according to roadsigns or roadtype) 3: S_ApproachOnMyLane : speed to high with respect to oncoming vehicle on my lane 4: S_Overtaken : I am being overtaken and I'm accelerating 6: S_RedTrafficLight : Should stop for red traffic light 7: S_YellowTrafficLight : should stop because of yellow traffic light

		8: S_EmergLeft : Vehicle from left takes unjust right of way, you should stop in order to avoid accident 9: S_RowLeft: should slow down to give vehicle from left right of way 10: S_EmergRight : Vehicle from right takes unjust right of way, you should stop in order to avoid accident 11: S_RowRight : should slow down to give vehicle from left right of way 12: S_EmergStraight : Vehicle from ahead takes unjust right of way, you should stop in order to avoid accident 13: S_RowStraight : should slow down to give vehicle from ahead right of way 14: S_AdaptInCurve : speed too high for curvature (while driving in curve) 15: S_AdaptApproachCurve: speed too high for curvature (while approaching curve) 16: S_AdaptInInterCurve : speed too high for curvature (while driving on intersection and turning left or right) 17: S_AdaptApproachInterCurve : speed too high for curvature (while approaching intersection and wanting to turning left or right)
d_WheelError	36	The deviation of the frontwheel angles from the normative wheel angle (given an ideal path). The deviation between the actual frontwheel angle and the roadgeometry (normative angle), in degrees (0..360). If on straight road the normative angle is the angle of the straight segment. If on a curved segment, the normative angle is the tangent of the angle of the line from the centerpoint of the arc to the coordinates of the center of the front bumper.
d_Tlc	37	Time-to-line crossing to the lane edgeline. This is the geometrically accurate version. If vehicle is moving to left edgeline: positive. If vehicle is moving to right edgeline: negative.
d_SteerError	38	The deviation between the actual steeringwheel angle and the required steeringwheel angle, in degrees. Required steeringwheel angle is computed from vehicle longitudinal velocity, yaw rate and frontwheel slipangles. Requires an accurate vehicle dynamics model, like the internal model of StTraffic
d_Thw	39	Time headway, in seconds, between first leadvehicle in same lane as MainTarget and MainTarget. Computed as distance/velocity (in m/s) of

		MainTarget. Distance is computed as distance along path between rearbumper of leadvehicle and frontbumper of MainTarget. Infinit is velocity of MainTarget = 0.
d_Ttc	40	Time-to-collision, in seconds, between first leadvehicle in same lane as MainTarget and MainTarget. Computed as distance/relative velocity (in m/s) of MainTarget. Distance is computed as distance along path between rearbumper of leadvehicle and frontbumper of MainTarget. Infinit is velocity of MainTarget = 0. Relative velocity is velocity of MainTarget – velocity of lead vehicle. If relative velocity <= 0, then TTC is computed as Infinit.
d_Tti	41	Time-to-intersection, in seconds. Computed as distance along path (between front bumper of MainTarget and start of intersection plane) to intersection/velocity of MainTarget (m/s). Infinit is velocity is 0.
d_TtcOpp	42	Time-to-collision between frontbumpers of MainTarget and a vehicle coming from opposite direction in same lane as MainTarget. Computed as distance along path/(velocity of MainTarget + velocity of oncoming vehicle).
d_StopDis	43	Current stopping distance, computed as – $Velocity^2$/(2*acc). Defined only if acc (acceleration) <= -0.1. Else the value is Infinit.
d_LeadDis	44	Bumper to bumper distance (in meters) along path to first leadvehicle in same lane as MainTarget
d_LeadVel	45	Velocity (in m/s) of first leadvehicle in same lane as MainTarget
d_RearDis	46	Bumper to bumper distance (in meters) along the path to first rearvehicle. Rearvehicle is not necessarily in same lane as MainTarget.
d_RearVel	47	Velocity (in m/s) of first rearvehicle. Rearvehicle is not necessarily in same lane as MainTarget.
d_ApprDis	48	Bumper to bumper distance (in meters) along the path to first approaching vehicle from opporite direction (on same road as MainTarget). Approaching vehicle is not necessarily in same lane as MainTarget.
d_ApprVel	49	Velocity (in m/s) of approaching vehicle from opporite direction (on same road as MainTarget). Approaching vehicle is not necessarily in same lane as MainTarget.
d_LeftDis	50	Distance to intersection (in meters to start of

		intersection plane) of first vehicle approaching the first oncoming intersection from left.
d_LeftVel	51	Velocity (in m/s) of first vehicle approaching the first oncoming intersection from left.
d_RightDis	52	Distance to intersection (in meters to start of intersection plane) of first vehicle approaching the first oncoming intersection from right.
d_RightVel	53	Velocity (in m/s) of first vehicle approaching the first oncoming intersection from right.
d_AheadDis	54	Distance to intersection (in meters to start of intersection plane) of first vehicle approaching the first oncoming intersection from opposite direction (with respect to MainTarget).
d_AheadVel	55	Velocity (in m/s) of first vehicle approaching the first oncoming intersection from opposite direction (with respect to MainTarget).
d_SteerSpeed	56	steeringwheel rotation velocity in degrees/second
d_Yawrate	57	Yawrate (rotationspeed of vehicle longitudinal axis)
d_Tlc_1	58	Crude approximation of TLC with respect to the lane edgelines. If moving to the right edgeline: Negative, computed as distance between front right of MainTarget and right edgeline/lateral velocity. If moving to the left edgeline: Positive, computed as distance between front left of MainTarget and left edgeline/ lateral velocity.
d_Latacc	59	Lateral acceleration computed as yawrate * longitudinal velocity of MainTarget

1.5 AddDataFunction

Proc(AddDataFunction, "*UserDefinedFunctionName"*);

Make sure that the name of the user function is entered between string symbols ("name").
The UserDefined Function must be defined before using the AddDataFunction procedure. Furthermore:

1) The UserDefined function must not have any inputparameters
2) The UserDefined function must always return a value

For example:

```
Define Function FuelConsumption() {
   FuelConsumption := fuelflow();   // fuel consumption in liters/minute
}
....
....
```

```
Proc( AddDataFunction, “FuelConsumption” );
```

This will result in sampling the fuelconsumption of the MainTarget (the car you are driving in). By this you can sample and store any variable you wish, as long as you define a function that returns the variable.

UserDefined function values are always stored in the record AFTER the variables defined by AddDataVariable.

For example:

```
Proc( SetSampleFrequency, 10 );
Proc( ClearDataVariables );
Proc( AddDataVariable, d_velocity );
Proc( AddDataFunction, FuelConsumption);
Proc( AddDataVariable, d_latpos );
Proc( AddDataVariable, d_gear );
```

results in the following dataformat for each datarecord:
<timestamp velocity latpos gear FuelConsumption>

1.6 Create and close a datafile

1.6.1 Create data file

Only one binary datafile can be opened for storing data at the same time. A datafile is created and openen by the procedure OpenData:

Proc(OpenData, string1, string2);

string1 contains the binary file name without the extention. The system adds the extention ‘da0’ to the filename.
string2 contains a text string that is included in the header of the datafile.

For example:

Proc(OpenData, ‘’subject01”, “experiment X12”);

This procedure does the following things:

- it creates a binary datafile with name ‘subject01.da0’, with the recordstructure defined by the AddDataVariable and AddDataFunction procedures. If the file already exists, it is overwritten.
- The text “experiment X12” is stored in the header of the file
- from that moment on data sampling is started with the frequency specified by the procedure SetSampleFrequency, until the file is closed (see next paragraph)
- it creates an eventfile with the name ‘subject01.evt’, in which discrete event can be stored that are time-locked to the binary data stream

The name of the file can be extracted in two ways:

1. Via the Subjects startup file that was created with ExpPrep. This file has an extention *.exp and consists of three lines:
    a. The name of the data analysis filename. If the experiment has been started from the *.exp file, the name can be retrieved via the SubjectIdent() function
    b. The name of the data definition (*.dd) file
    c. The name of the script (*.scb) file.
2. If a file named subjectfile.txt is located in the \data folder under the folder where the script is located, this file is read and the name in the file is used as a file name.

Examples of both methods are given here.

1.6.2 Subject startup file

If the experiment has been started via an *.exp file, then the function ExpDataDefined() returns the value true. In that case the datafile name can be read from script with the SubjectIdent() function, see next example.

```
Var { Sample; }
String { D_BaseFName; }
….
Sample := ExpDataDefined();
If ( Sample = True ) {
   D_BaseFName := SubjectIdent();
   Proc( OpenData, D_BaseFName, “Experiment 1”);
}
```

This method is used is you want to create subject startup files with ExpPrep, and then start the simulation/experiment via the <Start Simulation> button in control.exe. Starting a simulation with the <Start Simulation> button allows you to start the following file types:

- *.exp file
- *.scb file
- *.pac file

If you want to store driving errors in the *.xls spreadsheets as well, then this method does not work. In that case you need to use the next method (subjectfile.txt) to store data.

1.6.3 Subjectfile.txt

The alternative method is to have a file named ‘subjectfile.txt’ in the \data folder.
For example (from DataProc.sci):

```
If ( Sample = False ) {
   Proc( OpenFileRead, "subjectfile.txt");
   D_BaseFName := ReadString( "subjectfile.txt");
   Proc( CloseFile, "subjectfile.txt");
   ….
   Proc( OpenData, D_BaseFName, “Experiment 1”);
}
```

This method can be used together with storing data in the student assessment system (excel spreadsheets). The disadvantage of this method is that subjects.txt contains only 1 string with a file name, for example “P01A” (which results in p01A.da0 and p01a.evt). Before each and every experimental trial (subject) you need to modify the name then, which is inpractical and error prone. You can solve this by creating batch files (*.bat) to startup the experiment for every subject/trial combination. Suppose you have 5 subjects:

- create (in a text editor) 5 files, named p01-subject.txt, p02-subject.txt, p02-subject.txt, p04-subject.txt, p05-subject.txt. All these contain a different name, p01-subject.txt contains the value p01, p02-subject.txt contains the value p02 etc.These files must be located in the \data folder below the folder where the script is located.
- create (in a text editor) 5 different batch names, named p01.bat, p02.bat, p03.bat, p04.bat and p04.bat. These files must be located in the \data folder below the folder where the script is located. For example, p01.bat consists of:
  - copy p01-subject.txt subjectfile.txt
  - cd c:\DriveSim3\Carnetsoft\SimCarnet
  - control.exe
  - select the script from <Start Simulation>, OR start the simulation from <Student data> and select the file p01 (actually p01.xls, that you have created before) and then select the simulation form the spreadsheet.

If you don’t want to store data with his method anymore, make sure to delete or rename “subjectfile.txt”, for example as “subjectfile__.txt”

1.6.4 Close data file

Data sampling continues on the background of Traffic.exe until the procedure CloseData is called as:

Proc(CloseData);

From that moment on:

- Data sampling is terminated
- The binary datafile (extention .da0) is flushed and closed
- The event file (extention .evt) is flushed and closed

If you don’t use the CloseData procedure explicitly in our script, the files are flushed and closed on termination of the experiment. This is when the user applies the <Stop Simulation> button in Control.exe.

While the system is sampling data, all data are temporarily stored in a databuffer in the internal memory of the computer, until the buffer is full. Then the buffers are flushed and stored to external memory as a large datablock.

1.7 Set event codes

Together with the binary datafile, an eventfile is created with the same name as the binary datafile, but with extention .evt. The user is responsible for filling the eventfiles with data.

For data analysis it is often convenient to store events in an eventfile. You can then, for example, indicate when a certain scenario 20 starts and indicate the end of scenario 20 with the code 201 or whatever code you like. This facilitates the selection of datablocks during data analysis. Events can also be stored if something specific occurs, for example to signal when a lead vehicle starts to brake. If you then want to analyze the driver's brake response to this, the time-relation between brake pedal values and this event can be analyzed immediately, if you also store the brake pedal signal.

To store events in the eventfile, the following procedure is applied:

Proc(SetEventCode, <number>);

as in Proc(SetEventCode, 20);

Suppose you want to indicate the start of a scenario and the end of a scenario with eventcodes:

```
Define Scen[20] {
   Start {
          When ( Part[MainTarget].PathNr = 40 );
          Proc( SetEventCode, 20 );
          ….
    }
   End {
          When ( Part[MainTarget].PathNr = 54 );
          Proc( SetEventCode, 201 );
   }
}
```

The eventfile is a plain ASCII file with the following structure:

<Header, containing the text entered in Proc(OpenData, …, …) >
eventcode timestamp
eventcode timestamp
etc.

For example:

```
experiment X12
   20    10.030
   21    16.701
  120    24.030
   22    26.503
  121    34.893
   23    39.843
```

The timestamp is the time since the present simulation started and is synchronized with the timestamp values in the binary data file. Timestamp is equal to the system clocktime.

If you want to sample eventdata with external time references, such as heartrate data, you may want to use and alternative version:

Proc(SetTimeAndEventCode, <number>, <external clock value>);

Suppose you sample R-top values together with a timevalue on an external computer. These values can then be send to the present simulation with WriteUdp and ReadUdp functions. Upon arrival of a heartrate value, these values can be stored with the SetTimeAndEventCode procedure, while other useful events, like button presses or scenarionumbers can be stored in-between.

1.8. Dataformat binary datafiles

The binary datafiles start with a header of 2048 bytes. The number of (the sum of) datavariables and userdefined functions must be <= 32. So you cannot sample more than 32 variables at the same time.

The header consists of the following:

```
struct DataHeader { // 2 kb header block
   // first 1k block
   char ident[32];                  // contains the value 'DataProc'
   int version ;                    // contains the version: 2
   int subversion;            // contains the subversion: 0
   int datasize;                    // not used
   int nrfields;                    // number of variables used (sum of datavariables
                                    // and userfunctions: maximally 32
   int targetid;                    // allways 0: MainTarget
   char targetname[32];             // contains the string "Cabin car"
   float sampleinterval;            // sample frequency
   int  storagemode;                // 1 = raw values without time interpolation:
                                    //       if SampleFrequency has been set to 0 by user
                                    // 0 = time-interpolated sampling with frequency set by user:
                                    //       if frequency has been set by user to another value
                                    //       than 0. This is the normal operating mode.
   char filename[64];               // datafilename as specified by user
   char storedate[32];              // date and time of data storage
   char text1[128];                 // the text set by the user
   char text2[128];                 // not used

   // second 1k block
   char fieldnames[32][32];         // record definition: names of datavariables and functions
                                    // in sequential order
};
```

After the header there is a sequence of datarecords with the following structure:
timestamp <sizeof(float)>
followed by the values of all datavariables, in order of addition, <sizeof(float) > except:

d_LaneDirection	<sizeof(char)>
d_traflight	<sizeof(char)>

d_gear	<sizeof(short)>
d_indicator	<sizeof(short)>
d_segnum	<sizeof(short)>
d_pathnum	<sizeof(short)>
d_internum	<sizeof(short)>
d_scennum	<sizeof(short)>
d_LowestSpeedCause	<sizeof(short)>

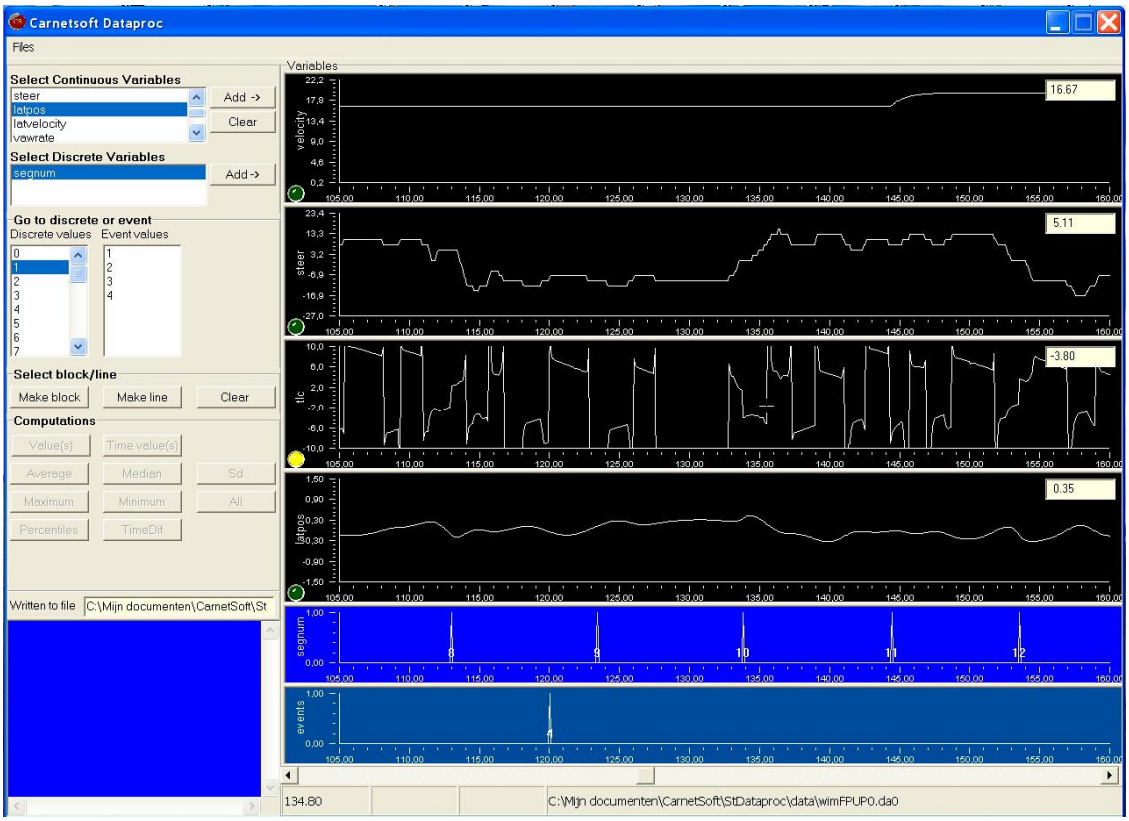
followed by the values of all data functions, in order of addition <sizeof(float) >

2. Data analysis using DataProc.exe

2.1 Introduction

For visual inspection of the data and data analysis the Dataproc program is available. The binary datafiles with extention .da0 are read by Dataproc, together with the eventfile and variables can be selected, visualized and analyzed. The analyzed data are written to an ASCII file with keywords to indicate the meaning of the data.

The following figure shows a snapshot of Dataproc, with vehicle speed, steering wheel angle, lateral position and tlc.



2.2 Starting Dataproc

Dataproc is started by clicking on the Dataproc icon on the desktop. The first thing to do is open a datafile.

2.3 Opening a datafile

To open a datafile, select <files> from the mainmenu and then <Open …>

Then a fileselection box pops up where all files with the extention *.da0 are listed.

Click on a file, and then on the button <Open>. The file the is opened and a list of all continuous variables is shown in the <Select Continuous Variables> listbox. All discrete variables are shown in the <Select Discrete Variables> listbox, see the following figure.

The program treats all variables and data functions as continuous except the following:
- d_segnum Segment number
- d_pathnum Path number
- d_scennum Scenario number
- d_internum Intersection number
- d_traflight Traffic light status

These are discrete variables that have no computational meaning, but serve as indicators for navigating through the file.

In this case, a scrollable list of all continuous variables that were sampled in the experiment is shown. In this example, there was only one discrete variable sampled: the segment number. This can be used to distinguish all road segments traversed during the experiment.
Furthermore, a list of event values is shown. These are all event codes that were registered during the experiment. If you click on one of these values, the file is repositioned from that event. In the lowest (lightblue) graph, all these events are shown as vertical lines with the event code next to it.

Now, continuous and discrete variables can be selected. Click on a variable in the continuous variables list and press the <Add> button. Up to 5 different continuous variables can be added to the charts and only one discrete variable can be added. After clicking on the <Add> button, a graph is added to the charts, for the continuous variables. If you select and add a discrete variable the following happens:
- the scrollable list <Discrete values> is filled with all sequential values of the discrete variable. In this case, all segmentnumbers are shown as they were sequentially traversed.
- The blue graph is filled with the values of the added discrete variable, indicated by vertical lines that signal the start of the repective number, together wit the value of the discrete variable.

2.4 Visualization of data

Up to 5 continuous variables can be visualized simultaneously. To select other variables, press the <Clear> button next to the list of continuous variables, and add new variables. The file can be scrolled by the horizontal scrollbar at the bottom of the window. Or you can select a discrete value or an eventcode and jump to that position in the file.

If you position the mouse pointer in any of the graphs, the value of the respective variables is displayed in the textboxes in the upper right corner of each graph. The timestamp is then displayed in the lower-left panel.

When all data are read from the input file, all graphs are scaled between the lowest and the highest values that occur within the file (with some exceptions, like the Tlc because these have infinite values as lowest or highest values). The user can change the scale of the Y-axis as follows:
- position the mouse cursor in the respective graph
- press the right mouse button. A small menu occurs, that allows to to change the range of the Y- axis. Press this.
- a popup window is displayed that shows the present Y axis maximum, its minimum, the number of main ticks and the number of subticks.
- Fill out the values that you want and press the <Ok> button. Now the grah is redrawn with the new Y-axis scale.

If you want the original values back, press the <Set default values> button.

2.5 Dataprocessing

Dataprocessing can be done in two ways:
- Make a vertical line and compute the value of the variable at that X position
- Make a block, consisting of a startline and an endline and compute measures on all values within that block.

To make a line do the following:
- Click on the <Make line> button
- The position the mouse cursor in any graph and click the left mouse button.
- The led light at the left side of the graph turn into yellow, to indicate that this variable has been selected
- Select the variable you want to measure by clicking in the appropriate graph
- Press the <Value(s)> button if you want to know the value of that variable at that point in time, or click the <Time value(s)> button to write the accompanying timestamp to the output file

These values are displayed in the blue messagebox at the lower-left side of the window. An outputfile is ALWAYS created. The name of the outputfile is displayed above the blue messagebox. The default filename is the name of the inputfile with the extention .out. If you process the same files more than once, the older output files are then overwritten. Because of that, it is recommended to change the name of the outputfile (see later).

To make a block, do the following:
- Click on the <Make block> button
- position the mouse cursor in any graph and click the left mouse button to create the start of the block
- position the mouse cursor in any graph and click the left mouse button to create the end of the block: two vertical lines now indicate the block
- Select the variable you want to measure by clicking in the appropriate graph
- The block consists of a vector of values. You can make the following computations on this vector, indicated by pressing the respective button:
  - Average: gives the mean value
  - Median : gives the median value
  - Sd : gives the standard deviation
  - Maximum : gives the highest value

- Minimum : gives the lowest value
- All : prints all values to the output file
- Percentiles: gives a list of percentiles, for positive and negative values separately: (10^{th}, 20^{th}, 30^{th}, 40^{th}, 50^{th}, 60^{th}, 70^{th}, 80^{th} and 90^{th} percentile values)
- TimeDif : the length of the block in seconds.

2.6 Other functions

2.6.1 Changing the name of the outputfile

To change the name of the outputfile do the following:
- in the main menu, select <Files> followed by <Save as…>.
- a file selection box appears in which you can make a new filename. Only non-existing filenames are accepted. The extention ".out" is added by the program.

The only purpose of this is to change the name of the outputfile with all processed data. All data are saved automatically as you process the data, so you never need to save the file explicitly.

2.6.2 Converting the binary datafile to an ascii file with column headers

To export the file to an ascii format output do the following:
- in the main menu, select <Files> followed by <Convert to ascii …>
- a file selection box appears in which you can make a new filename. The extention ".dat" is added by the program.

The ascii files can be processed by other programs like Excel.

3. Processed Data via script

Processed data are data that are measured, analyzed and stored DURING the simulation. After the experiment trial, you have these data available which can save a lot of time spent on data processing.
DataProc.sci contains some functionality for this, and it's easy to extend this in script.

In this method, you create an ascii file (using script) and fill it with the appropriate values.

All examples are derived from DataProc.sci.

The function to open a file:

```
Define Function ClearLookModes() {
   LastLookDir := 0;
   NrLookLeft := 0;
   NrLookRight := 0;
   DurLookLeft := 0.0;
   DurLookRight := 0.0;
}

Define Function ClearVariables() {
   NrSamples     := 0;
   SumSpeed      := 0;
   SquareSpeed := 0;
   AvgSpeed      := 9999;
   SdSpeed       := 9999;

   SumLatpos     := 0;
   SquareLatpos:= 0;
   AvgLatpos     := 9999;
   SdLatpos      := 9999;
   NrCollisions := 0;
}

Define Function WriteHeader() {
   TextLine := "Path Avg_Sp  SD_Sp Avg_Lp SD_Lp  NrColl NrLeft NrRight DurLeft DurRight";
   Proc( WriteFile, D_FileName, TextLine );
}

Define Function StartDataProcessing() {
   Var { a; Sample; }
   Sample := False;
   If ( D_DataProcessingOn = False ) {
      a := ClearVariables();
      a := ClearLookModes();
      BlockCounter     := 0;
      // Open file, try method 1 first
      Sample := ExpDataDefined();
      If ( Sample = True ) {
         D_FileName := SubjectIdent();
         D_FileName := strcat( D_FileName, ".txt");
```

```
      }
      If ( Sample = False ) { // if method 1 failed, then try method 2
        Proc( OpenFileRead, "subjectfile.txt");
        D_FileName := ReadString( "subjectfile.txt");
        Proc( CloseFile, "subjectfile.txt");
        If ( strlen( D_FileName ) > 0 ) {
           Sample := True;
           D_FileName := strcat( D_FileName, ".txt");
        }
      }
      If ( Sample = True ) {
        Proc( OpenFile, D_FileName );
        D_DataProcessingOn := True;
        Proc( Print, "Ascii datafile opened...." );
        a := WriteHeader();
      }
    }
    StartDataProcessing := Sample;
}

Define Function WriteAsciiData( PathNumber ) {
    Var { a; temp; }
    If ( D_DataProcessingOn = True ) {
       If ( NrSamples > 2 ) {
         temp := (SquareSpeed-(sqr(SumSpeed)/NrSamples))/(NrSamples-1);
         If ( temp <= 0 ) { SdSpeed := 0.0; }
         Else { SdSpeed := sqrt(temp); }
         AvgSpeed := SumSpeed/NrSamples;
         temp := (SquareLatpos-(sqr(SumLatpos)/NrSamples))/(NrSamples-1);
         If ( temp <= 0 ) { SdLatpos := 0.0; }
         Else { SdLatpos := sqrt(temp); }
         AvgLatpos := SumLatpos/NrSamples;
       }
       BlockCounter := BlockCounter + 1;
       TextLine := "";
       TextLine := strcat( TextLine, num2str( BlockCounter, 4, 0 ));
       TextLine := "";
       TextLine := strcat( TextLine, num2str( PathNumber, 4, 0 ));
       TextLine := strcat( TextLine, " ");
       TextLine := strcat( TextLine, num2str( 3.6*AvgSpeed, 10, 4 ));
       TextLine := strcat( TextLine, " ");
       TextLine := strcat( TextLine, num2str( 3.6*SdSpeed, 10, 4 ));
       TextLine := strcat( TextLine, " ");
       TextLine := strcat( TextLine, num2str( AvgLatpos, 10, 4 ));
       TextLine := strcat( TextLine, " ");
       TextLine := strcat( TextLine, num2str( SdLatpos, 10, 4 ));
       TextLine := strcat( TextLine, " ");
       TextLine := strcat( TextLine, num2str( NrCollisions, 4, 0 ));
       TextLine := strcat( TextLine, " ");
       TextLine := strcat( TextLine, num2str( NrLookLeft, 3, 0 ));
       TextLine := strcat( TextLine, " ");
       TextLine := strcat( TextLine, num2str( NrLookRight, 3, 0 ));
       TextLine := strcat( TextLine, " ");
       TextLine := strcat( TextLine, num2str( DurLookLeft, 5, 2 ));
       TextLine := strcat( TextLine, " ");
```

```
      TextLine := strcat( TextLine, num2str( DurLookRight, 5, 2 ));
      Proc( WriteFile, D_FileName,TextLine );

      a := ClearVariables();
      a := ClearLookModes();
      Proc( Print, "Write to ascii datafile ...." );
   }
}
```

Then start this by adding in a script:

```
   Define Action[0] {
     Start {
        When ( StartRun = True );  // when experiment starts, start data processing
        Action[].NrTimes := 1;
        ProcessData := StartDataProcessing();
        D_DataProcessingOn := False; // don't measure data during the first path
     }
   }
   Define Action[1] {
     Start {
        When ( Stopped = True );  // when experiment stops, stop data processing
        If ( ProcessData = True ) {
           a := StopDataProcessing();
        }
     }
   }
   Define Action[2] {
     Start {
        When ( ProcessData = True and Part[MainTarget].PathNr != LastPath and
                Part[MainTarget].DisFromInter > 10 ); // when you arrive at a new path, start measuring
        LastPath := Part[MainTarget].PathNr;
        D_DataProcessingOn := True;   // flag to control start measurement
     }
     End {
        When ( Part[MainTarget].PathNr != LastPath ); // and write to file when a new path is entered
        a := WriteAsciiData( LastPath ); // write data
        D_DataProcessingOn := False;  // stop measurement
     }
   }

   Define Action[3] {
     Start {
       When ( D_DataProcessingOn = True );    // start data processing
       a := ClearVariables();
       a := ClearLookModes();
     }
     Do {  // compate data for SD speed and SD lateral position
       ThisTime := runtime();
       If ( (ThisTime - LastTime) > 0.1 ) {
         NrSamples := NrSamples + 1;
         SquareSpeed := SquareSpeed + sqr(Part[MainTarget].Velocity);
         SumSpeed    := SumSpeed + Part[MainTarget].Velocity;
         latpos := Part[MainTarget].LatPos + 1.5;
         SquareLatpos := SquareLatpos + sqr(latpos);
```

```
        SumLatpos  := SumLatpos + latpos;
        LastTime := ThisTime;
      }
    }
    End {
      When ( D_DataProcessingOn = False );
    }
  }

  Define Action[4] {
    Var { StartTime; EndTime; }
    Start {
      When ( TestLookDir != LastLookDir );
      LastLookDir := TestLookDir;
      StartTime := runtime();
      If ( LastLookDir = LOOK_LEFT ) {
        NrLookLeft := NrLookLeft + 1;
      }
      ElseIf ( LastLookDir = LOOK_RIGHT ) {
        NrLookRight := NrLookRight + 1;
      }
    }
    End {
      When ( TestLookDir != LastLookDir );
      EndTime := runtime();
      If ( LastLookDir = LOOK_LEFT ) {
        DurLookLeft := DurLookLeft + (EndTime-StartTime);
      }
      ElseIf ( LastLookDir = LOOK_RIGHT ) {
        DurLookRight := DurLookRight + (EndTime-StartTime);
      }
    }
  }
```

4. Driving errors in Student Assessment System

The highest level type of data that can be stored consists of driver errors that are measured as part of the student assessment system in the driver training software modules. This is explained in detail in CarnetManualEN30.pdf.

The SAS is the central part of the simulator training. Every student has a spreadsheet (Excel spreadsheet) with the results stored. Each student has a separate spreadsheet to store the progress data. When a new student starts the training, a new spreadsheet has to be created for that student. These data can be used in the research simulator as well and can be useful when the 'quality of driving' has to be assessed. There are around 40 'lessons' available, ranging from simulated highway driving to driving in rural areas, on roundabouts and in villages and towns. Also night driving, driving in slippery road conditions or in fog is simulated.

These simulations can be modified by the user by changing and recompiling the scripts. Normally, a virtual instructor gives feedback on driver errors, but this can be disabled via IOConfig.exe. The scripts also give instructions via graphical popups, which can be disabled in the script.

For every subject, a new SAS spreadsheet has to be created. This can be done in two ways:
- Click on the NewStudent desktop icon, and add a new subject (names 'student' in this application).
- All spreadsheets are located in c:\DriveSim3\Carnetsoft\SimCarnet\Students. Open this folder and copy LVSblanco.tem to <subjectname>.xls, for example:
  copy LVSblanco.tem p01.xls

Every spreadsheet has a tab for each lesson, and each lesson has 10 columns to store data, so a specific lesson can be stored 10 times. This is enough for most experiments where the number of within-subject repeated measurements is less than 10.

To start an experiment using the SAS data, while measuring raw data plus processed data, do the following:
- start the simulator from the batch files, as described in paragraph 1.6.3, for example p01.bat for subject p01.
- press <Student data > button.
- press <Select student> button, select the correct subject, for example p01 (which is p01.xls)
- if the database has been loaded (takes a while), select the proper script and press <Start Lesson>
- After the simulation (or lesson as it is called here) is finished, the data are stored in the spreadsheet, and the raw data plus processed data are stored in the \data folder within the folder where the selected 'lesson' is located.