Extending a Model with Embedded Python

This technical memo shows by examples how a VehicleSim (VS) vehicle math model is extended using the embedded Python interpreter.

VS Solvers have internal math models with built-in equations used to predict the behavior of vehicles in response to the driving environment and driver/rider controls. In addition to these “native” models, VehicleSim products provide numerous resources to the user to extend these models or to provide alternative models completely. These resources are described in the tech memo: *Automating Runs with the VS API*, and the tech memo: *Extending VS Math Models with VS Commands and the VS API*.

The above-mentioned resources can make use of external simulation environments such as Simulink (MATLAB) or LabVIEW as platforms for the user to build their own models. In addition, the user can make use of the VS API to build or enhance math models in various compiled languages, including C, C++, and Visual Basic. The VS API can also be accessed via an external Python interpreter.

The VS API allows external code to control the simulation by accessing and advancing the simulation by a time step. Between time steps, external code can access and modify variables of the simulation and then advance the simulation further. In this way, an advanced user can modify or replace a portion of the simulation model and still make use of the remaining portions of the simulation to effect a desired result.

An advanced user can also extend models with VS Commands. This is done within the solver framework and needs no external resources. Variables are modified between time steps via runtime equations applied at various times within the simulation (`EQ_INIT`, `EQ_IN`, `EQ_DYN`, etc.). However, VS Commands have limits. They cannot handle arrays; they cannot loop; they do not support user-defined functions; etc.

To support more complicated model extensions made within the solver, VS Solvers now support an internal (embedded) Python interpreter.

Note	Currently, Embedded Python cannot be used for Real Time system applications.

The Python Interpreter

The Python interpreter is accessed by one of two commands. There is the VS Command:

```
RUN_PYTHON_STRING “string”
```

With this command, a string is directly processed by the Python interpreter. The main use for this command is to designate a Python module for use by the simulation, e.g.:

```
RUN_PYTHON_STRING “import steercontrol”
```

In this case, the module `steercontrol.py` is loaded into the interpreter, and its functions and variables are available for use. The interpreter looks in a few subdirectories to access module files. The preferred directory for the user to locate their Python modules is `xxx_Data\Extensions\Python`.]

The model functions themselves are accessed with another command, `Python()`. Inputs and outputs for the function calls are accomplished with user-defined VS tables. An example call is shown:

```
EQ_IN pyret = PYTHON(pycond,"steercontrol.pre","Op",in_table,out_table)
```

With this call, several elements are present, which will be explained. First, this equation is an expression applied with the `EQ_IN` command, which means the equation is applied at: “every time step, just before the built-in equations of motion are applied.” The `PYTHON()` call can be applied with any of the other equations commands (`EQ_INIT`, `EQ_DYN`, `EQ_OUT`, etc.).

The `pyret` variable is used to receive the result of the Python call. The result is primarily to indicate the success or failure of code operation itself; output relating to the simulation variables themselves are sent to the output array. Having the `PYTHON()` command return a value also puts the call in the form of an equation, which the equation commands are expecting.

The `PYTHON()` call has five parameters. The first parameter, `pycond`, is a conditional that determines whether the call is executed. If the value is 0, then the call is not run. If it is non-zero, then the call will be run. The conditional parameter is of value if the Python call executes a very complicated procedure which need not be updated at every time step.

The second parameter is the Python function to be run. This is usually in the form of a string `"module.function"`, based on a module which has been previously imported. This references the function that performs the desired operation(s) in Python. Since the function can take multiple inputs and produce multiple returns, a single function may access and update several VS variables.

The third parameter is also a string (needing quotes) which is used as an optional text input to the Python function. This is the signal string and can be used to provide a readable conditional input to the function, or a text parameter (such as a filename) that may be needed by the function. The parameter does not need to be used and can be ignored by the function.

The fourth parameter is a table of type `STEP` which is used to provide inputs to the Python function. The table is defined previously, but updated VS variables can be loaded into the array (using the appropriate `EQ_XX` command). The array needs to be one dimensional (a `TABLE`, not a `CARPET`) and large enough to hold all the inputs needed by the function.

The fifth and last parameter is also a table of type `STEP` which is used to hold the outputs from the Python function. Like the inputs, the outputs are floating point values. As with the input table, the output table should be sized for the outputs produced. After calling the Python function, the output array values can then be used to update VS variables using the add equation commands (`EQ_XXX`). If the keyword `None` is provided as either the fourth or fifth parameter (or both) then either the input or output is ignored. The command `define_table` is limited to creating tables of size two or greater.

If the `EQ_XXX` command format is not desired, a call can be made as a standalone VS Command as well:

```
        RUN_PYTHON_PROG module.function signal in_table out_table
```

This functions similarly to the `Python()` except there is no return value, and no conditional parameter that determines whether the call is executed. Other than that, this access to the Python utility is similar to the `Python()` call.

Here are some VS commands that demonstrate how the variable arrays are defined and loaded, the Python command is called, and how the output values are retrieved.

```
! Define table for inputs to Embedded Python
define_table in
in_table step
1,1
2,2
3,3
4,4
endtable

! Define table for outputs from Embedded Python
define_table out
out_table step
1,1
```

```
2,2
endtable

! Define 3 new parameters.
define_parameter L_FORWARD 20; m; Distance to view point
define_parameter LAT_TRACK -1.6; m; Offset from road centerline
define_parameter GAIN_STEER_CTRL 10; deg/m; Control gain

! Use pycond to execute Python code intermittently if desired.
define_variable pycond = 1
define_variable tcount = 0
eq_in pycond = if_gt_0_then(FMOD(tcount,50), 0, 1)
eq_in tcount = tcount+1

! Assign variables and parameters to input table
eq_in in(0,1,1) = L_FORWARD
eq_in in(0,2,1) = YAW
eq_in in(0,3,1) = XCG_TM
eq_in in(0,4,1) = YCG_TM

! Run the routine to calculate X and Y preview values from Yaw
! and vehicle CofG
eq_in pyret = python(pycond,"steercontrol.pre","Op",in_table,out_table)
eq_in xprev = out(0,1,1)
eq_in yprev = out(0,2,1)
```

In the above code segment, the parameter `L_FORWARD` and the variables `YAW`, `XCG_TM`, and `YCG_TM` are used to calculate the new variables `xprev` and `yprev`. Because these calls make use of an add equation command (`EQ_XX`) the equations examined at every time step. In this example, however, the conditional `pycond` is set such that the Python code is executed once every 50 time steps.

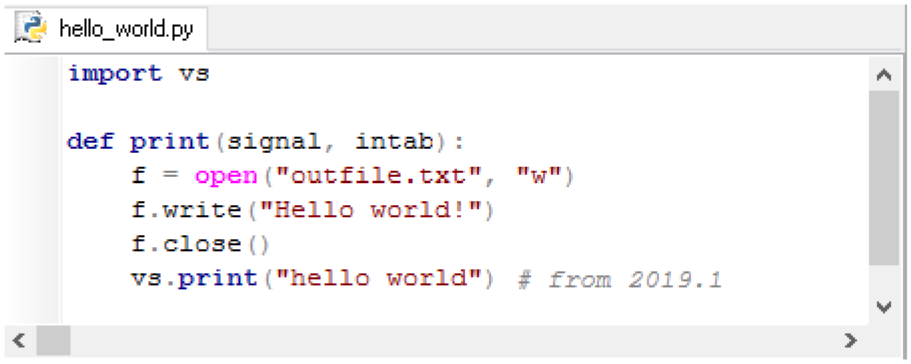
Example 1: Hello World

The intent of this example is to demonstrate how to call a simple Python script and print a message during the simulation. Even though the example includes a vehicle and a procedure, it does not utilize them. The example contains a 'hello world' VS commands dataset which can be accessed from the **Run Control** screen. Since there is no parameter exchange between VehicleSim product (such as CarSim) and the Python script, only three VS commands are required.

```
OPT_ENABLE_PYTHON = 1

RUN_PYTHON_STRING "import hello_world"

RUN_PYTHON_PROG hello_world.print 'OPEN' None None
```

The embedded Python option is not enabled by default, so it needs to be explicitly enabled by OPT_ENABLE_PYTHON command. To load a module, use RUN_PYTHON_STRING. It imports **hello_world.py** which resides in `xxx_Data\Extensions\Python`. **hello_world.py** contains a function called `print`, and the function can be accessed by RUN_PYTHON_PROG command. In this example, the second, the third, and the fourth parameters are not being used. RUN_PYTHON_PROG executes only once.

The contents of **hello_world.py** is shown in Figure 1. This script simply writes the string “Hello World” in a file called outfile.txt and also in a pop-up window. After running the simulation, outfile.txt can be found in `xxx_Data` directory.



```
import vs

def print(signal, intab):
    f = open("outfile.txt", "w")
    f.write("Hello world!")
    f.close()
    vs.print("hello world") # from 2019.1
```

Figure 1. Hello World.

Note A listing of VS Python functions available for use with the Embedded Python utility can be found the Embedded Python section of the “VS Commands Reference Manual”.

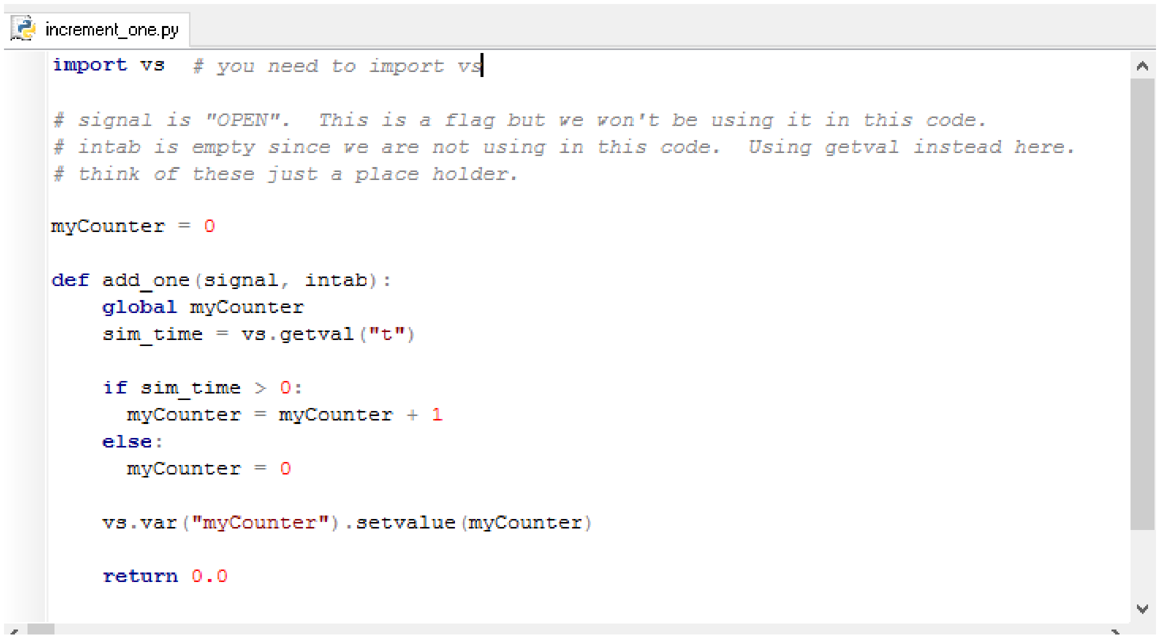
Example 2: Add One at Each Time Step

The intent of this example is to demonstrate how to call a Python script at every time step. This example also shows how values can be obtained and updated directly, without having to access the input or output tables. As in Example 1, this example has vehicle and procedure datasets but it does not utilize them. The VS Commands dataset called ‘increment one’ is accessed from the **Run Control** screen and it imports and execute a Python script called **increment_one.py**. Since the purpose of this script is to add a value 1 to a variable at every time step, it uses `python()` command with the `EQ_IN`.

```
eq_in pyReturn = python(pyCondition,"increment_one.add_one","OPEN",None,None)
```

With this command, a function, `add_one`, in **increment_one.py** is called. In the above command, the third parameter, “OPEN”, the fourth, and the fifth parameters are not.

increment_one.py is included in `xxx_Data\Extensions\Python` (Figure 2). The script receives a time value from a VehicleSim product using `vs.getval()` and adds 1 to `myCounter` if time is more than 0. `myCounter` is defined as `global` so that it retains the value throughout the simulation, not just each time step. At each call, using `vs.var().setvalue()` command, `myCounter` which is defined in the Python script assigns the value to `myCounter` which is previously defined in the VS Commands dataset.


increment_one.py

```
import vs   # you need to import vs

# signal is "OPEN".  This is a flag but we won't be using it in this code.
# intab is empty since we are not using in this code.  Using getval instead here.
# think of these just a place holder.

myCounter = 0

def add_one(signal, intab):
    global myCounter
    sim_time = vs.getval("t")

    if sim_time > 0:
      myCounter = myCounter + 1
    else:
      myCounter = 0

    vs.var("myCounter").setvalue(myCounter)

    return 0.0
```

Figure 2. Script to Add Value 1 to a Counter at Every Time Step.

Example 3: Simple Braking

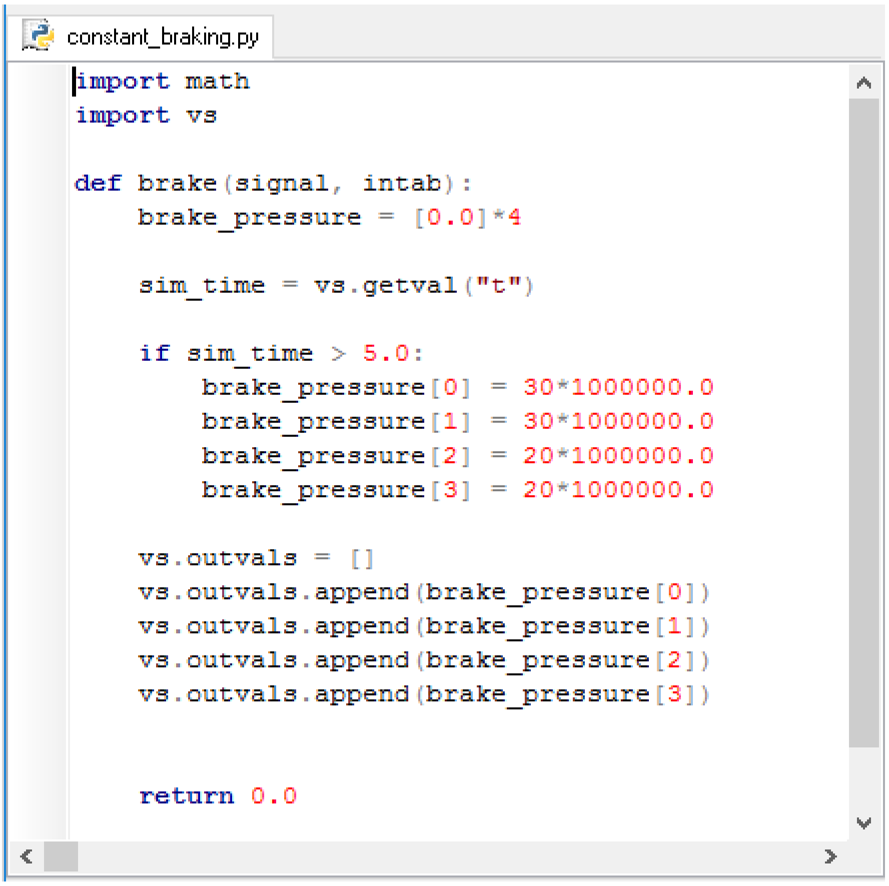
The idea of this Simple Braking embedded python example is to introduce users how to use an output table. What this example does is to run a vehicle at a target speed and apply brake actuator pressure at each wheel when the simulation time is greater than 5.0 seconds.

As with Examples 1 and 2, there is a VS Commands dataset which is accessible from the **Run Control** screen. In the dataset, the embedded Python option is enabled, and a Python script called **constant_braking.py** is imported and executed. In **constant_braking.py**, 'time' value is obtained using `vs.getval()`, and when time is larger than 5.0 seconds, brake actuator pressures are applied and return these pressure values to a Vehiclesim product. In Example 2, `vs.var().setvalue()` command is used to set value of VS variable, `myCounter`. However, here, a table is used for output which stores and update values. It is possible not to use a table and just use `vs.var().setvalue()`, but since a table acts as an array, it looks more organized especially if there are many inputs and/or outputs. A table type STEP is used for output and is defined in VS commands dataset.

```
! Define table for outputs from Embedded Python
define_table outtab
out_table step
1,1
2,2
3,3
4,4
Endtable
```

Since four actuator brake pressure values are going to be updated every time step, the table is sized for four outputs. It is important to match a table size and output numbers. The function brake in **constant_braking.py** is called using python(). There is no input table for this call. The outputs are stored in outtab_table.

```
eq_in pyReturn = python(pyCondition, "constant_braking.brake", "OPEN", None,
outtab_table)
```



```
import math
import vs

def brake(signal, intab):
    brake_pressure = [0.0]*4

    sim_time = vs.getval("t")

    if sim_time > 5.0:
        brake_pressure[0] = 30*1000000.0
        brake_pressure[1] = 30*1000000.0
        brake_pressure[2] = 20*1000000.0
        brake_pressure[3] = 20*1000000.0

    vs.outvals = []
    vs.outvals.append(brake_pressure[0])
    vs.outvals.append(brake_pressure[1])
    vs.outvals.append(brake_pressure[2])
    vs.outvals.append(brake_pressure[3])


    return 0.0
```

Figure 3. Apply Brake Actuator Pressures.

In constant_braking.py (Figure 3), the values are assigned to brake_pressure[0] to [4]. The script accesses to the output list vs.outvals and append those brake actuator pressure values into outtab table. After python() call is made from VS Commands dataset, these pressure values are assigned to IMP_PBK_X.

```
eq_in IMP_PBK_L1 = outtab(0,1,1)
eq_in IMP_PBK_R1 = outtab(0,2,1)
eq_in IMP_PBK_L2 = outtab(0,3,1)
eq_in IMP_PBK_R2 = outtab(0,4,1)
```

Note This example uses a solver wrapper to execute the simulation. This is not required for this example, but some packages (such as NumPy) may require embedded Python simulations to be executed with a solver wrapper.

Example 4: Simple Steering Control

A simple steering control will be used as an example model extension. This is the same controller that has been used in several other examples, including the example using VS commands. The controller is based on a “preview point” in front of the vehicle and the relationship of that preview point to the road centerline. If the point is not on target, a steering wheel angle is calculated that is proportional to the lateral distance between the point and the target lateral position. Figure 4 shows the point represented with an arrow in front of the vehicle. The intended location is the center of the right-hand lane. The arrow is a little to the right of the target line, and therefore the vehicle steering wheel should be turned to the left.

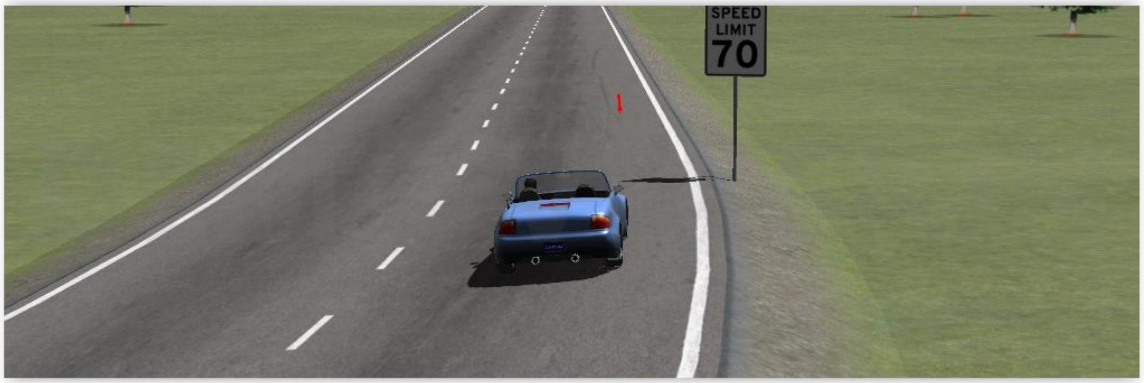


Figure 4. Vehicle with a preview point for a simple steering controller.

Note The example controller presented in this memo is described more fully in another memo “Extending VS Math Models with VS Commands and the VS API”, which be reviewed for more details of this model extension. This section will focus on the implementation of this model extension using the embedded Python option.

Existing Model Documentation

The full list of available Import variables for a VS model is generated by using the **View** button in the lower-right corner of the **Run Control** screen of a product such as CarSim. Two versions are available: one for viewing with a text editor, and one for viewing as a spreadsheet.

If you browse the documentation file for any of the vehicle models in CarSim or TruckSim, you will see that they all include steering wheel angle as a variable that can be imported.

The default settings for the Import variables are that they be ignored. The imported steering wheel angle can be activated by providing a line of text in any data field that provides input to the model:

```
IMPORT IMP_STEER_SW VS_REPLACE
```
(1)

Although the native equations in the VS Solver include the effect of the imported steer, they do not define the angle. (After all, the whole purpose of the imported steer is to replace or modify the value already available in the native equations of the model.) Therefore, the example extensions will calculate the steering wheel angle, based on the position of the target point, which in turn depends on the current vehicle position.

The output variables available in a VS Math Model can also be viewed using the **View** button on the **Run Control** screen. Reading through the list of available variables shows that there are many X and Y coordinates, for points such as the origin of the sprung mass coordinate system, the vehicle center of gravity (CG), etc. A preview point for a new steering controller can be based on any of the existing X and Y coordinate pairs. In this example, the X and Y coordinates of the vehicle CG will be used.

Defining the Steering Controller

The Steering Controller is defined with the following sequence of commands:

The imported steering wheel angle is activated by providing a line of text in any data field that provides input to the model:

```
IMPORT IMP_STEER_SW VS_REPLACE                                  (1)
```

The X and Y coordinates of a preview point are defined with:

```
DEFINE_OUTPUT XPREVIEW = XCG_TM + L_FORWARD*COS(YAW); m; X Coord  (2)
DEFINE_OUTPUT YPREVIEW = YCG_TM + L_FORWARD*SIN(YAW); m; Y Coord  (3)
```

where

`XCG_TM` and `YCG_TM` are output variables that provide the X and Y coordinates of the center of mass of the vehicle,

`YAW` is the output variable for the yaw angle of the vehicle, and

`L_FORWARD` is a parameter for the distance the preview point lies in front of the vehicle mass center.

The desired steer angle during the run will be:

```
STEER_CTRL = (LAT_TRACK - ROAD_L(XPREVIEW, YPREVIEW))*GAIN_STEER_CTRL (4)
```

where

`LAT_TRACK` is the target lateral position relative to the road centerline,

`ROAD_L` is a function available in VS Solvers (see VS Commands) that gives the lateral distance of a point defined by X and Y coordinates relative to the road reference line (typically the centerline),

`XPREVIEW` and `YPREVIEW` are X and Y coordinates of a point in front of the vehicle, and

`GAIN_STEER_CTRL` is a constant used to scale the steering wheel angle.

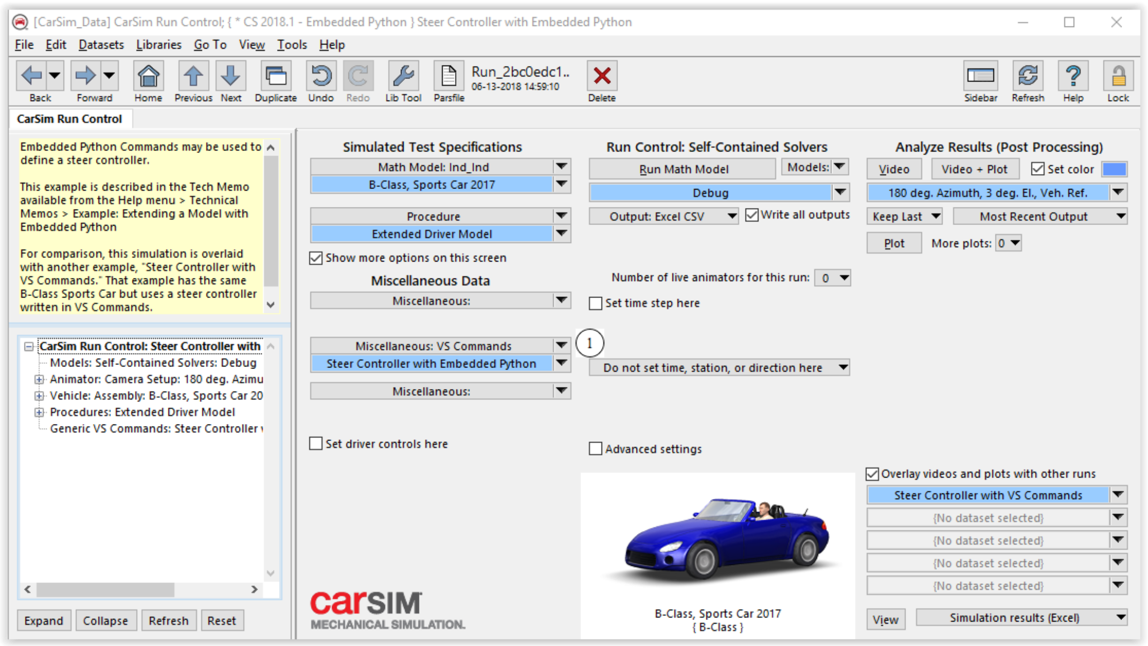
The controller calculation is potentially complicated at the start of the run because not all of the output variables have been calculated yet. For this simple example, the steering wheel angle will be defined as zero until the simulation time is greater than zero. Therefore, equation 2 is replaced with a more complicated expression:

```
IMP_STEER_SW = IF_GT_0_THEN(T,
   (LAT_TRACK -ROAD_L(XPREVIEW, YPREVIEW))*GAIN_STEER_CTRL, 0)  (5)
```

This sets the steer to 0 until T is greater than 0, and then uses the expression from equation 2.

Programming the Steer Controller in Python

This section describes how the steer controller is added to a math model using Embedded Python calls. The run screen for the Embedded Python Controller is shown below:



[CarSim_Data] CarSim Run Control; { * CS 2018.1 - Embedded Python } Steer Controller with Embedded Python

Run_2bc0edc1..
06-13-2018 14:59:10

CarSim Run Control

Embedded Python Commands may be used to define a steer controller.

This example is described in the Tech Memo available from the Help menu > Technical Memos > Example: Extending a Model with Embedded Python

For comparison, this simulation is overlaid with another example, "Steer Controller with VS Commands." That example has the same B-Class Sports Car but uses a steer controller written in VS Commands.

CarSim Run Control: Steer Controller with
Models: Self-Contained Solvers: Debug
Animator: Camera Setup: 180 deg. Azimu
Vehicle: Assembly: B-Class, Sports Car 20
Procedures: Extended Driver Model
Generic VS Commands: Steer Controller

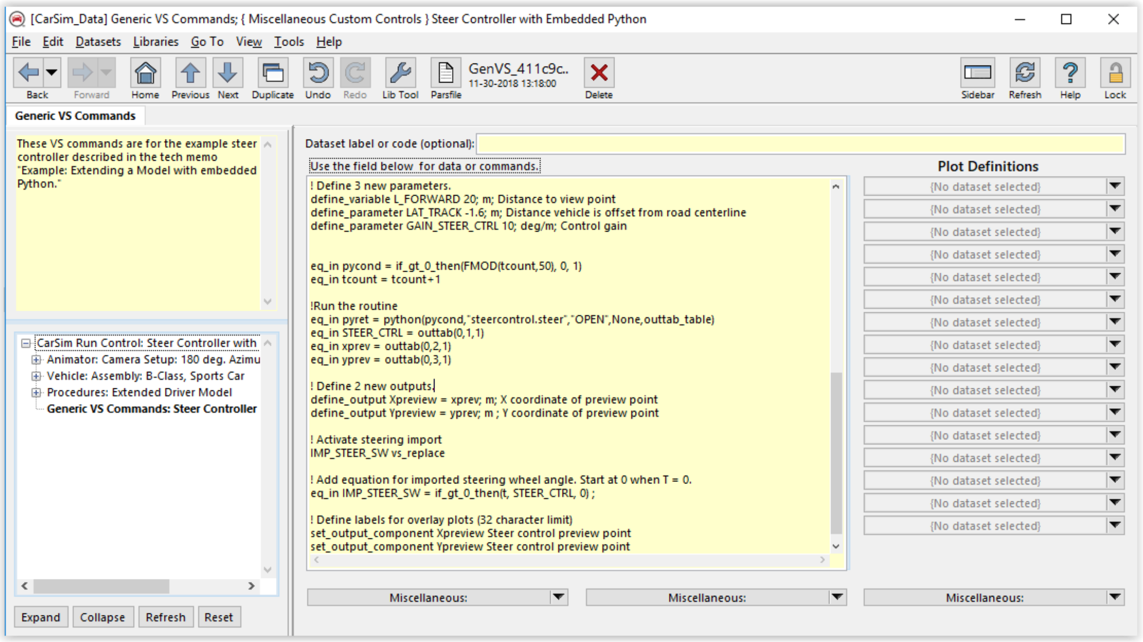
Simulated Test Specifications
Math Model: Ind_Ind
B-Class, Sports Car 2017
Procedure
Extended Driver Model
☑ Show more options on this screen
Miscellaneous Data
Miscellaneous:
Miscellaneous: VS Commands
Steer Controller with Embedded Python
Miscellaneous:
☐ Set driver controls here

Run Control: Self-Contained Solvers
Run Math Model
Models:
Debug
Output: Excel CSV
☑ Write all outputs
Number of live animators for this run: 0
☐ Set time step here
1
Do not set time, station, or direction here
☐ Advanced settings
B-Class, Sports Car 2017
{ B-Class }

Analyze Results (Post Processing)
Video
Video + Plot
☑ Set color
180 deg. Azimuth, 3 deg. El., Veh. Ref.
Keep Last
Most Recent Output
Plot
More plots: 0
☑ Overlay videos and plots with other runs
Steer Controller with VS Commands
{No dataset selected}
{No dataset selected}
{No dataset selected}
{No dataset selected}
View
Simulation results (Excel)

Figure 5. Run Screen for a simple steering controller using Embedded Python.

This screen is very similar to the Run Screen used for defining the Steer Controller with VS Commands. The primary difference is the VS Commands Screen at 1, ‘Steer Controller with Embedded Python’. So let’s look at that:


[CarSim_Data] Generic VS Commands; { Miscellaneous Custom Controls } Steer Controller with Embedded Python

Generic VS Commands

These VS commands are for the example steer controller described in the tech memo "Example: Extending a Model with embedded Python."

- CarSim Run Control: Steer Controller with
  - Animator: Camera Setup: 180 deg. Azimu
  - Vehicle: Assembly: B-Class, Sports Car
  - Procedures: Extended Driver Model
  - **Generic VS Commands: Steer Controller**

Dataset label or code (optional):

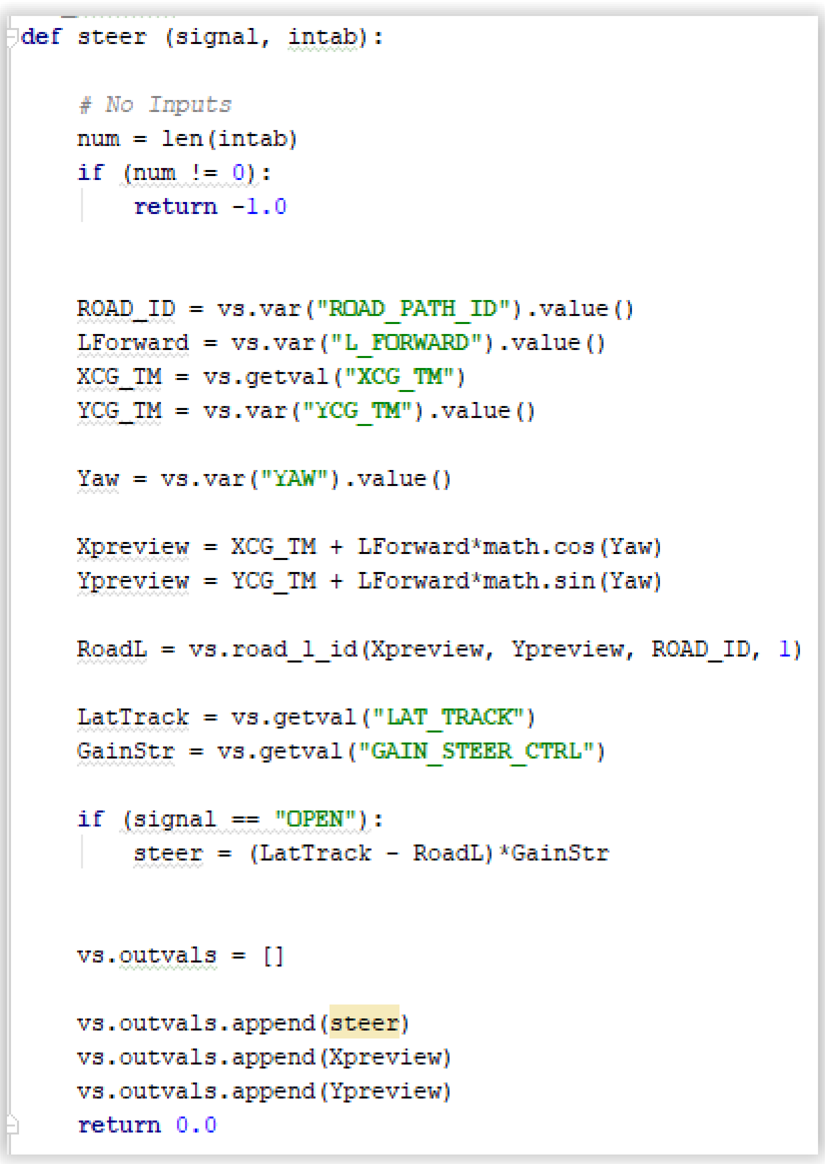
Use the field below for data or commands.

```
! Define 3 new parameters.
define_variable L_FORWARD 20; m; Distance to view point
define_parameter LAT_TRACK -1.6; m; Distance vehicle is offset from road centerline
define_parameter GAIN_STEER_CTRL 10; deg/m; Control gain


eq_in pycond = if_gt_0_then(FMOD(tcount,50), 0, 1)
eq_in tcount = tcount+1

!Run the routine
eq_in pyret = python(pycond,"steercontrol.steer","OPEN",None,outtab_table)
eq_in STEER_CTRL = outtab(0,1,1)
eq_in xprev = outtab(0,2,1)
eq_in yprev = outtab(0,3,1)

! Define 2 new outputs.
define_output Xpreview = xprev; m; X coordinate of preview point
define_output Ypreview = yprev; m ; Y coordinate of preview point

! Activate steering import
IMP_STEER_SW vs_replace

! Add equation for imported steering wheel angle. Start at 0 when T = 0.
eq_in IMP_STEER_SW = if_gt_0_then(t, STEER_CTRL, 0) ;

! Define labels for overlay plots (32 character limit)
set_output_component Xpreview Steer control preview point
set_output_component Ypreview Steer control preview point
```

Figure 6. VS Command Screen (upper) for a simple steering controller using Embedded Python.

In this first part of the VS Commands Screen, the LAT_TRACK and GAIN_STEER_CTRL parameters and the `L_FORWARD` variables are defined and given values. The `python` command is called (also at every time step, conditioned on the value of `pycond`) and the output placed in `outtab`. The variables STEER_CTRL, `xprev` and `yprev` are loaded from the output table, again at every time step.

The Python routine referenced and called with the access call is `steercontrol.steer`. This is routine in the Python file **steercontrol.py** and is shown below (Figure 7).



```
def steer (signal, intab):

    # No Inputs
    num = len(intab)
    if (num != 0):
        return -1.0


    ROAD_ID = vs.var("ROAD_PATH_ID").value()
    LForward = vs.var("L_FORWARD").value()
    XCG_TM = vs.getval("XCG_TM")
    YCG_TM = vs.var("YCG_TM").value()

    Yaw = vs.var("YAW").value()

    Xpreview = XCG_TM + LForward*math.cos(Yaw)
    Ypreview = YCG_TM + LForward*math.sin(Yaw)

    RoadL = vs.road_l_id(Xpreview, Ypreview, ROAD_ID, 1)

    LatTrack = vs.getval("LAT_TRACK")
    GainStr = vs.getval("GAIN_STEER_CTRL")

    if (signal == "OPEN"):
        steer = (LatTrack - RoadL)*GainStr


    vs.outvals = []

    vs.outvals.append(steer)
    vs.outvals.append(Xpreview)
    vs.outvals.append(Ypreview)
    return 0.0
```

Figure 7. Python routine 'steer' which generates Steer, Xpreview and Ypreview.

Note In order to use of the Embedded Python capability outside of the VS Browser (such as running a simulation with a C or C++ wrapper), make sure that the accessory files are available to the Embedded Python. The easiest way to do this, if you are running outside of the browser directory structure, is to copy directory trees `.\Programs\Python` and `.\Programs\Python\Lib` (and their contents) to the directory containing the executable (e.g., solver_simple.exe). Any Python files in `XXX_Data\Extensions\Python` needed for a simulation can be copied over to `.\Programs\Python`.

The Python routine, in this simple example, calculates the values for STEER_CTRL, `Xpreview` and `Ypreview`, which had been calculated natively with the VS Commands implementation of this controller. The difference is that if a more complicated calculation is needed, the user has access to the full Python language (loops, different data types, etc.) as well as the Python libraries of calls and routines. The input values used by the Python routine are either are accessed directly with VS specific calls that can access VS parameters and values (as in this case) or via an input list that can be passed into the function in the form of a VS STEP table. In this case, the needed inputs are loaded into the table, and the Python routine accesses the values in the form of a Python list of real values. For this call, the signal value (a string) and the name of the output table (also a string) are used. The 'NONE' value indicates that an input table is not used. The outputs produced by the Python routine use the `vs.outvals` variable (accessed from the vs module) to hold the output values to be sent back to the VS simulation.

`ROAD_L_ID()` is a VS command which is also accessible from the Embedded Python. After Xpreview and Ypreview have been calculated, as with the VS Commands version of the Steer Control, ROAD_L_ID() is called to get the lateral offset from the road for the point given to it. This value, along with other values retrieved (in this example) using vs.getval() to get the present value of a VS parameter or variable are used to calculate the steer value.

The routine uses the inputs to calculate the steer value using the simple equation that was previously accomplished with VS Commands. In this routine, the `signal` value used to use the "OPEN" steer equation. If other steer equations were to be considered, they could be accessed with other `signal` values. After the Python routine is called, the steer value (`STEER_CTRL`) is applied to `IMP_STEER_SW` to complete the model. `Xpreview` and `Ypreview` are also available to allow the drawing of the preview arrow in the video simulation.

The Python routine file itself needs to be located somewhere where the Embedded Python can find it. At present (2019.1) the Embedded Python looks for directories relative to the directory location of the executable being run (for example, ..\CarSim_Prog or ..\TruckSim_Prog). The directories sought out include XXX_Prog\Programs\Python, XXX_Prog\Programs\Python\Lib, and XXX_Data\Extensions\Python. The 'Lib' directory is part of the Embedded Python system and needs to be accessible for the Python routines to run.

This is a simple implementation of a steering controller which makes use of the embedded Python utility. It demonstrates how data is transferred to and from the Python environment.

Example 5: Changing a Target Path

This section describes another example which uses the Embedded Python to develop a simple collision avoidance simulation. This example is also interesting because it demonstrates how the Embedded Python is used to generate a custom table, which is not possible to do with regular VS Commands.

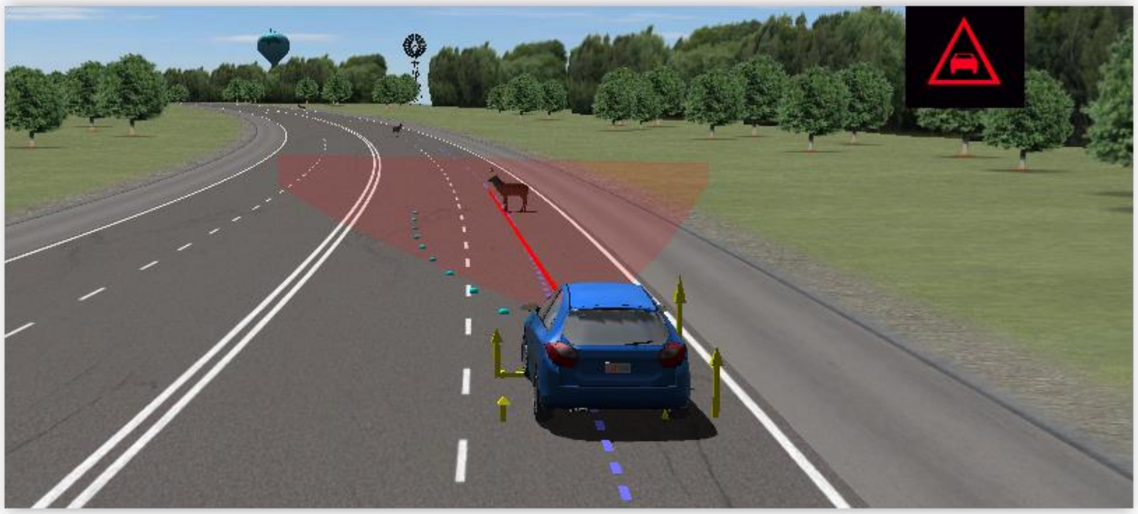


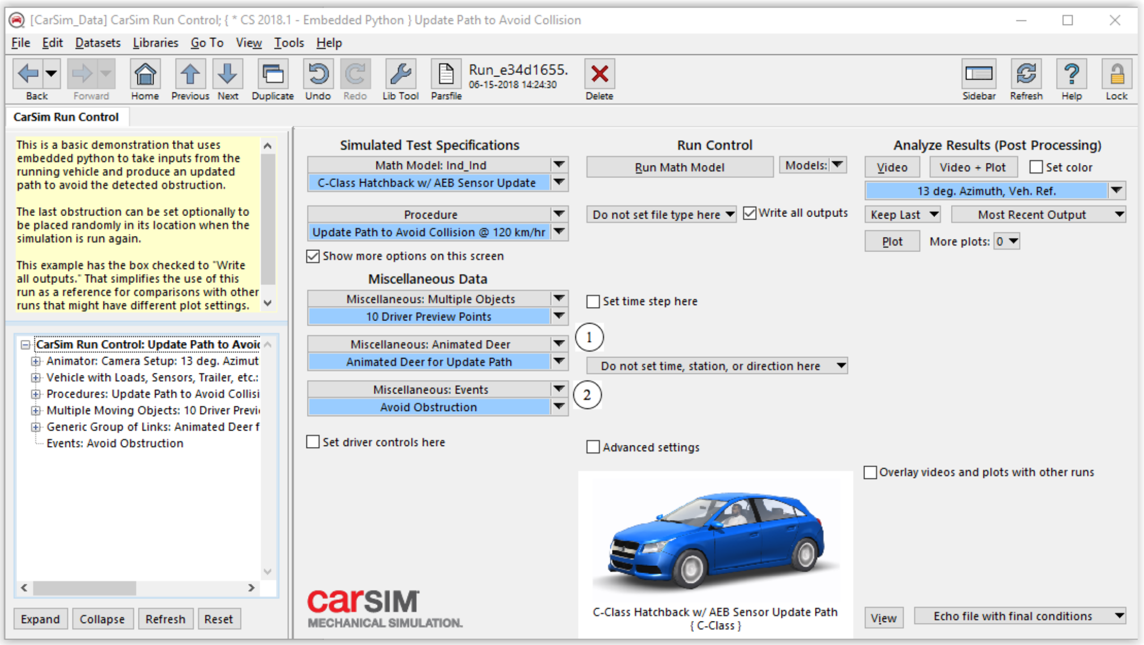
Figure 8. Simulation of vehicle that generates new paths to avoid obstacles.

Overview of the Simulation

A vehicle is directed to follow a path on a slowly curving road. The path followed is designated with the table `LTARG`, which is merely a constant offset -1.6 meters from the road center. Several obstacles are scattered on the road. The vehicle detects the obstacles with a Forward Facing Sensor. Upon detecting an obstacle, the simulation progresses through a series of events to detect the object and produce a new path to avoid the object. This is done by creating a new path in the form of a stored table file. This table is then loaded into the simulation to replace the existing `LTARG` table. The simulation then proceeds, with a new path redirecting the vehicle around the obstacle. The vehicle then proceeds as before, ready to avoid any further obstacles it may encounter.

This example, in addition to using the Embedded Python to plot an avoidance path, the VS commands `LOAD_TABLE_FILE` and `FILE_TO_TABLE` are used to update the `LTARG` table. Also, using the `SRAND()` and `GENSEED()` functions can be used to randomize the location of the final obstacle in the simulation. The obstacle will be randomly placed in a different location each time the simulation is re-run. To activate this randomization, a single line needs to be commented out on the Generic Group of Links page where the obstacles are defined and placed.

The run screen for the Collision Avoidance Simulation is shown below (Figure 9).


[CarSim_Data] CarSim Run Control; { * CS 2018.1 - Embedded Python } Update Path to Avoid Collision

Run_e34d1655. 06-15-2018 14:24:30

CarSim Run Control

This is a basic demonstration that uses embedded python to take inputs from the running vehicle and produce an updated path to avoid the detected obstruction.

The last obstruction can be set optionally to be placed randomly in its location when the simulation is run again.

This example has the box checked to "Write all outputs." That simplifies the use of this run as a reference for comparisons with other runs that might have different plot settings.

CarSim Run Control: Update Path to Avoi
- Animator: Camera Setup: 13 deg. Azimut
- Vehicle with Loads, Sensors, Trailer, etc.:
- Procedures: Update Path to Avoid Collisi
- Multiple Moving Objects: 10 Driver Previ
- Generic Group of Links: Animated Deer f
- Events: Avoid Obstruction

Simulated Test Specifications
- Math Model: Ind_Ind
- C-Class Hatchback w/ AEB Sensor Update
- Procedure
- Update Path to Avoid Collision @ 120 km/hr
- ☑ Show more options on this screen

Miscellaneous Data
- Miscellaneous: Multiple Objects
- 10 Driver Preview Points
- Miscellaneous: Animated Deer
- Animated Deer for Update Path
- Miscellaneous: Events
- Avoid Obstruction
- ☐ Set driver controls here

Run Control
- Run Math Model
- Models:
- Do not set file type here
- ☑ Write all outputs
- ☐ Set time step here
- Do not set time, station, or direction here
- ☐ Advanced settings

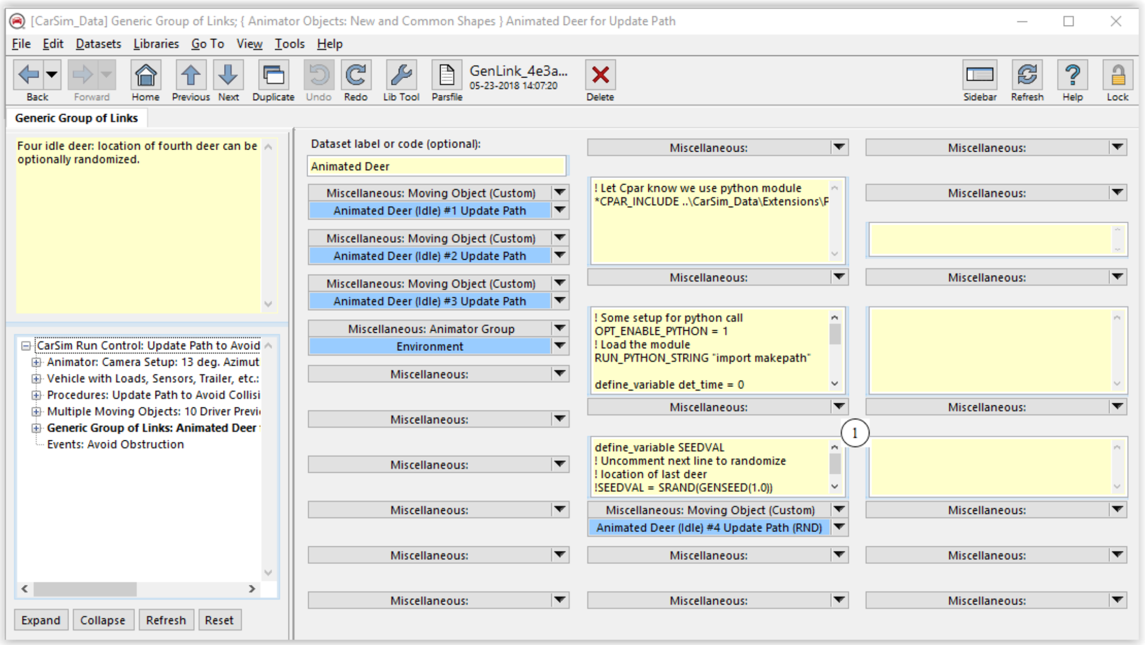
C-Class Hatchback w/ AEB Sensor Update Path { C-Class }

Analyze Results (Post Processing)
- Video
- Video + Plot
- ☐ Set color
- 13 deg. Azimuth, Veh. Ref.
- Keep Last
- Most Recent Output
- Plot
- More plots: 0
- ☐ Overlay videos and plots with other runs
- View
- Echo file with final conditions

File Edit Datasets Libraries Go To View Tools Help
Back Forward Home Previous Next Duplicate Undo Redo Lib Tool Parsfile Delete Sidebar Refresh Help Lock
Expand Collapse Refresh Reset
1
2
carSIM MECHANICAL SIMULATION.

Figure 9. Run Screen for collision avoidance simulation using Embedded Python.

The run screen has two links of particular note, (1) the link to the obstacles placed in the path and (2) the link to the events sequence to effect a path change to avoid an obstacle. The screen defining the obstacles is shown below (Figure 10).


[CarSim_Data] Generic Group of Links; { Animator Objects: New and Common Shapes } Animated Deer for Update Path

GenLink_4e3a... 05-23-2018 14:07:20

Generic Group of Links

Four idle deer: location of fourth deer can be optionally randomized.

CarSim Run Control: Update Path to Avoid
- Animator: Camera Setup: 13 deg. Azimut
- Vehicle with Loads, Sensors, Trailer, etc.:
- Procedures: Update Path to Avoid Collisi
- Multiple Moving Objects: 10 Driver Previ
- Generic Group of Links: Animated Deer
- Events: Avoid Obstruction

Dataset label or code (optional): Animated Deer

- Miscellaneous: Moving Object (Custom) — Animated Deer (Idle) #1 Update Path
- Miscellaneous: Moving Object (Custom) — Animated Deer (Idle) #2 Update Path
- Miscellaneous: Moving Object (Custom) — Animated Deer (Idle) #3 Update Path
- Miscellaneous: Animator Group — Environment

```
! Let Cpar know we use python module
*CPAR_INCLUDE ..\CarSim_Data\Extensions\F
```

```
! Some setup for python call
OPT_ENABLE_PYTHON = 1
! Load the module
RUN_PYTHON_STRING "import makepath"

define_variable det_time = 0
```

```
define_variable SEEDVAL
! Uncomment next line to randomize
! location of last deer
!SEEDVAL = SRAND(GENSEED(1.0))
```

- Miscellaneous: Moving Object (Custom) — Animated Deer (Idle) #4 Update Path (RND)

File Edit Datasets Libraries Go To View Tools Help
Back Forward Home Previous Next Duplicate Undo Redo Lib Tool Parsfile Delete Sidebar Refresh Help Lock
Miscellaneous:
Expand Collapse Refresh Reset
1

Figure 10. Obstacle Screen for collision avoidance simulation defining obstacles to be avoided.

In Figure 10, the yellow field at (1) potentially assigns a random value to `SEEDVAL`. But this is only done if the comment designation (`!`) is removed. If `SEEDVAL` is active, then the fourth obstacle moves randomly (within certain bounds) each time the simulation is run.

The sequence of events used by the collision avoidance simulation is summarized in Figure . The simulation begins with the vehicle awaiting a detection (Avoid Obstruction). After an obstacle has been detected, information is stored in the input array, awaiting the call to the Embedded Python (New Detect). This is repeated (information is overwritten with new information) until the distance from the obstacle drops below 35 meters. At this point, the Python routine is called with the loaded information, a new table is created, and this table is loaded to become the new LTARG table (Update path). At this point, the new LTARG table is used, and the vehicle follows the new path to avoid the obstacle. Time advances for 3 seconds and then it moves to the next event, awaiting a new detection (Returned to Path). Upon encountering a new obstacle, the event advances again, repeating the process (New Detect).

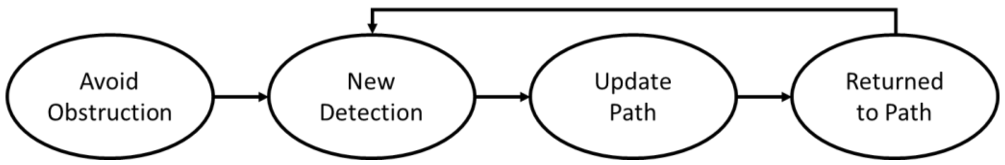


Figure 11. Event sequence for the collision avoidance simulation.

Generating a new path in Python

The Python routine used to generate the new path is shown in Listing 1. Using inputs reflecting the current state of the vehicle (`Velocity`, `Distance` and `Bearing` of Obstacle, `Lat_Targ`, and `Station`) five points are created which define a curved path which departs from the nominal path (a fixed offset from the road center), curves to avoid the obstacle, and then returns to the nominal path. These five points are defined as a list of two-valued entries (Station and Offset) which is then passed to `vs` module function `create_table`. The `signal` value is used to define the name of the table file to be created. The file is then loaded into the table `LTARG` using `LOAD_TABLE_FILE`. More sophisticated paths, perhaps accounting for the velocity of the vehicle and perhaps containing more than five points, could be developed and examined using this framework.

Listing 1. *Python routine used to create new path based on obstacle location.*

```python
import math
import vs
# This is a routine which takes input from a running vehicle and
# creates a new path to avoid the detected obstruction.
def pathtoavoid (signal, intab):
    vals = []
    entry = [0.0]*2

    #Inputs: Velocity, Dist., Bearing, Lat_Targ, Station
    num = len(intab)
    if (num < 5):
        return -1.0

    Velocity = intab[0]
    Distance = intab[1]
    Bearing =  intab[2]
    LatTarg =  intab[3]
    Station =  intab[4]

    #Normalize to nominal 120 km/hr = 33.33 m/s
    velnorm = 1.0 # For now, do not make use of velocity

    # How far off-center is the obstruction?
    offset = math.fabs(Distance*math.tan(Bearing))
    pathshift = (4.5 - offset)
    if (Bearing > 0.045):
        pathshift = -pathshift

    #define 5 points which can be used to define a Flat Spline
    entry[0] = (Station)
    entry[1] = (LatTarg)
    vals.append(entry.copy())

    entry[0] = (Station+(velnorm*Distance/2))
    entry[1] = (LatTarg+pathshift/math.sqrt(2))
    vals.append(entry.copy())

    entry[0] = (Station+Distance)
    entry[1] = (LatTarg+pathshift)
    vals.append(entry.copy())

    entry[0] = (Station+(3*velnorm*Distance)/2)
    entry[1] = (LatTarg+pathshift/math.sqrt(2))
    vals.append(entry.copy())

    entry[0] = (Station+(2*velnorm*Distance))
    entry[1] = LatTarg
    vals.append(entry.copy())

    #signal value is used to name path file
    vs.create_table(vals, signal, 'SPLINE_FLAT', 'm', 'm')

    return 0.0
```

The new table defining the avoiding path is created and loaded during the Update Path Event.

Note	When updating a table in VS, it is important to do so only in the context of a new event. Without a transition to a new event (or even returning to an event which has been visited previously) any table which is defined or loaded will only retain the final state for the simulation. Variables can be updated during a simulation with the `EQU_XXX` commands, but these commands cannot be used for tables. An event transition is needed to allow a table update to be used and recognized in a simulation.

Example 6: Update a Vehicle Path and Note Collisions

This section describes another example which uses the Embedded Python to update a vehicle path and note collisions. This example is similar to the previous example "Changing a Target Path" in that it also generates a custom table to create a new path. The example goes further in that the path is generated based on multiple detections, a recovery path can also be generated, and collisions can be detected and noted.

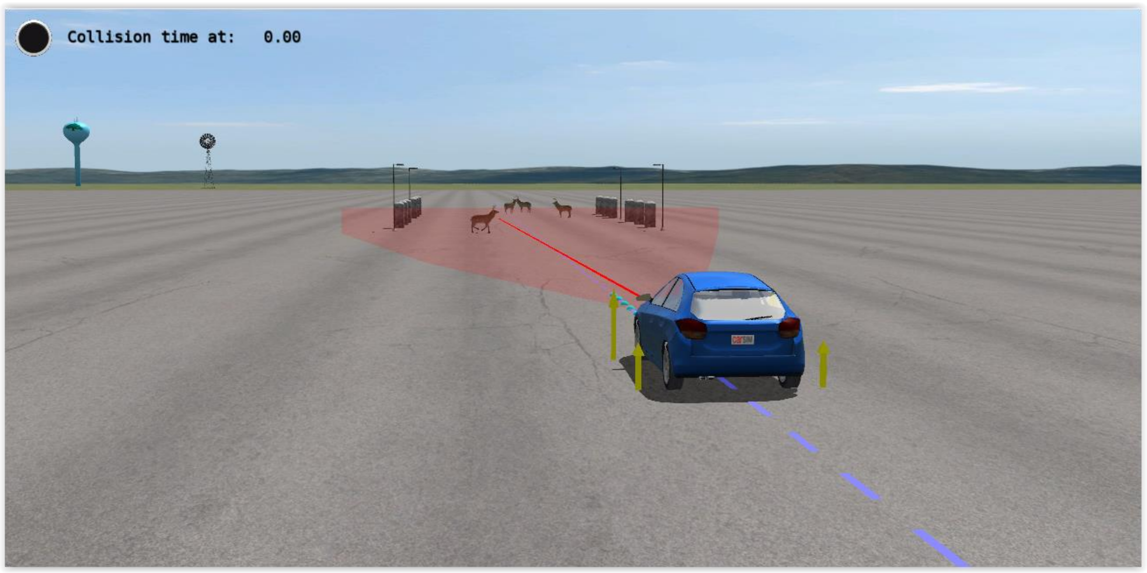


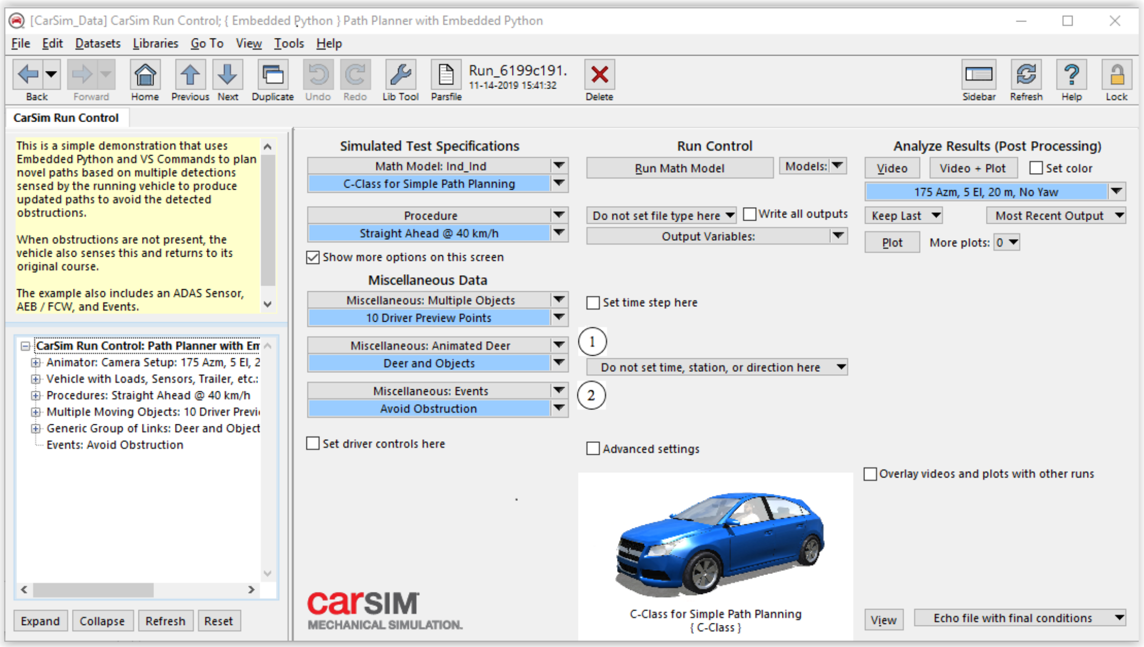
Figure 12. Simulation of vehicle that updates path to avoid multiple obstacles.

Overview of the Simulation

A vehicle is directed to follow a straight path with obstacles on both sides, and some obstacles (deer) on the path itself. The obstacles can be moving, and one of the deer is moving. The vehicle detects the obstacles with a Forward Facing Sensor. Upon detecting an obstacle, the simulation progresses through a series of events to detect the object(s) and produce a new path to avoid the object(s). This is done by creating a new path in the form of a stored table file. This table is then loaded into the simulation to replace the existing `LTARG` table. The simulation then proceeds, with a new path redirecting the vehicle around the obstacle. The vehicle then proceeds as before, ready to avoid any further obstacles it may encounter. If no obstacles are detected, and the vehicle finds itself off the desired path (due to previous path adjustments) it will readjust itself to get back on the desired path.

In addition to creating a new path based on detected obstacles, the simulation also detects any collisions the vehicle may have with any of the objects. If a collision occurs, then the red collision signal is turned on and the time of collision is noted. As configured, the vehicle does not collide with any objects. To get the vehicle to collide with an object, the position of the fourth deer can be adjusted so that the vehicle will collide with it. Instructions on how the position can be adjusted can be found in the yellow comments section of the obstacle screen (Figure 14).

The run screen for the Collision Avoidance Simulation is shown below (Figure 13).


[CarSim_Data] CarSim Run Control; { Embedded Python } Path Planner with Embedded Python

File Edit Datasets Libraries Go To View Tools Help

Run_6199c191.
11-14-2019 15:41:32

CarSim Run Control

This is a simple demonstration that uses Embedded Python and VS Commands to plan novel paths based on multiple detections sensed by the running vehicle to produce updated paths to avoid the detected obstructions.

When obstructions are not present, the vehicle also senses this and returns to its original course.

The example also includes an ADAS Sensor, AEB / FCW, and Events.

CarSim Run Control: Path Planner with Em
- Animator: Camera Setup: 175 Azm, 5 El, 2
- Vehicle with Loads, Sensors, Trailer, etc.:
- Procedures: Straight Ahead @ 40 km/h
- Multiple Moving Objects: 10 Driver Previ
- Generic Group of Links: Deer and Object
- Events: Avoid Obstruction

Simulated Test Specifications
Math Model: Ind_Ind
C-Class for Simple Path Planning
Procedure
Straight Ahead @ 40 km/h
[✓] Show more options on this screen

Miscellaneous Data
Miscellaneous: Multiple Objects
10 Driver Preview Points
Miscellaneous: Animated Deer
Deer and Objects
Miscellaneous: Events
Avoid Obstruction
[ ] Set driver controls here

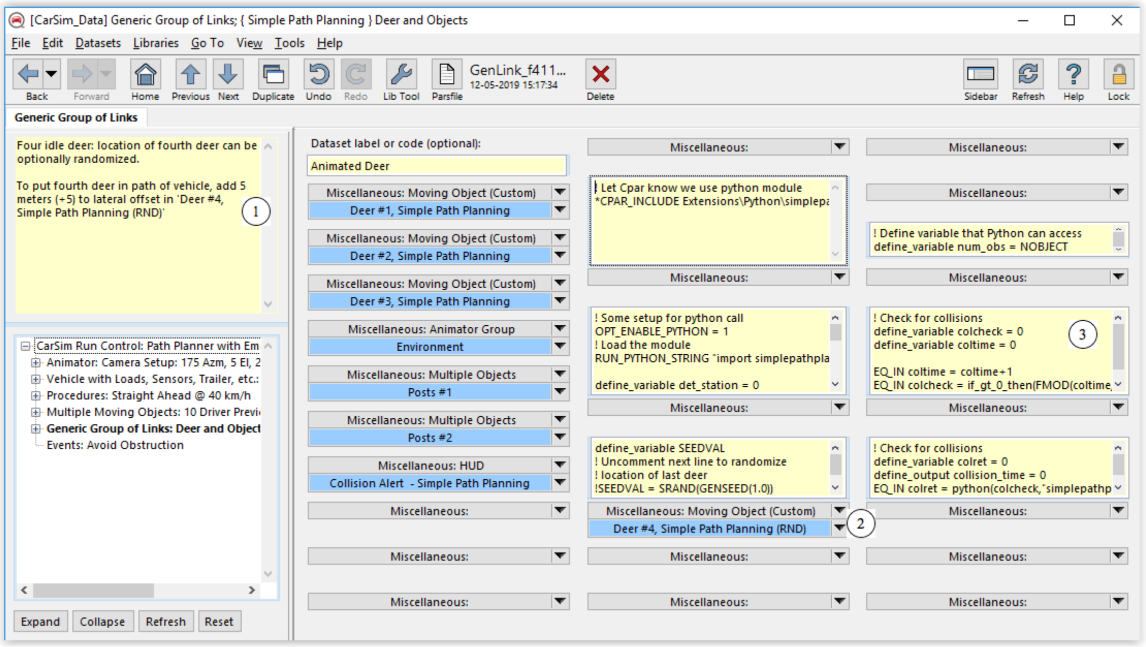
Run Control
Run Math Model
Models:
Do not set file type here
[ ] Write all outputs
Output Variables:
[ ] Set time step here
1
Do not set time, station, or direction here
2
[ ] Advanced settings

C-Class for Simple Path Planning
{ C-Class }

Analyze Results (Post Processing)
Video
Video + Plot
[ ] Set color
175 Azm, 5 El, 20 m, No Yaw
Keep Last
Most Recent Output
Plot
More plots: 0
[ ] Overlay videos and plots with other runs
View
Echo file with final conditions

Figure 13. Run Screen for collision avoidance simulation using Embedded Python.

The run screen has two links of particular note, (1) the link to the obstacles placed in the path and (2) the link to the events sequence to effect a path change to avoid an obstacle. The screen defining the obstacles is shown below (Figure 14).


[CarSim_Data] Generic Group of Links; { Simple Path Planning } Deer and Objects
File Edit Datasets Libraries Go To View Tools Help
Back Forward Home Previous Next Duplicate Undo Redo Lib Tool Parsfile
GenLink_f411...
12-05-2019 15:17:34
Delete
Sidebar Refresh Help Lock
Generic Group of Links
Expand Collapse Refresh Reset
1
2
3

Four idle deer: location of fourth deer can be optionally randomized.

To put fourth deer in path of vehicle, add 5 meters (+5) to lateral offset in 'Deer #4, Simple Path Planning (RND)'

- CarSim Run Control: Path Planner with Em
  - Animator: Camera Setup: 175 Azm, 5 El, 2
  - Vehicle with Loads, Sensors, Trailer, etc.:
  - Procedures: Straight Ahead @ 40 km/h
  - Multiple Moving Objects: 10 Driver Previ
  - **Generic Group of Links: Deer and Object**
  - Events: Avoid Obstruction

Dataset label or code (optional): Animated Deer

Miscellaneous: Moving Object (Custom) — Deer #1, Simple Path Planning
Miscellaneous: Moving Object (Custom) — Deer #2, Simple Path Planning
Miscellaneous: Moving Object (Custom) — Deer #3, Simple Path Planning
Miscellaneous: Animator Group — Environment
Miscellaneous: Multiple Objects — Posts #1
Miscellaneous: Multiple Objects — Posts #2
Miscellaneous: HUD — Collision Alert - Simple Path Planning
Miscellaneous: Moving Object (Custom) — Deer #4, Simple Path Planning (RND)

```
! Let Cpar know we use python module
*CPAR_INCLUDE Extensions\Python\simplepa
```

```
! Define variable that Python can access
define_variable num_obs = NOBJECT
```

```
! Some setup for python call
OPT_ENABLE_PYTHON = 1
! Load the module
RUN_PYTHON_STRING "import simplepathpla

define_variable det_station = 0
```

```
! Check for collisions
define_variable colcheck = 0
define_variable coltime = 0

EQ_IN coltime = coltime+1
EQ_IN colcheck = if_gt_0_then(FMOD(coltime,
```

```
define_variable SEEDVAL
! Uncomment next line to randomize
! location of last deer
!SEEDVAL = SRAND(GENSEED(1.0))
```

```
! Check for collisions
define_variable colret = 0
define_output collision_time = 0
EQ_IN colret = python(colcheck,"simplepathp
```

Figure 14. Obstacle Screen for path planning simulation defining obstacles to be avoided.

In Figure 14, the yellow field at ① indicates how to modify the position of Deer #4 so that a collision will occur. This is done by accessing the Blue Field at ②. The Yellow Field at ③ contains the commands for the simulation to check for collisions. The variable 'num_obs' (defined in the field above) is used by the Embedded Python to determine the number of objects in the simulation.

The sequence of events used by the collision avoidance simulation is summarized in Figure The simulation begins with the vehicle awaiting a detection (Avoid Obstruction). After an obstacle has been detected, information is stored in the input array, awaiting the call to the Embedded Python (New Detect). This is repeated (information is overwritten with new information) until the distance from the obstacle drops below 35 meters. At this point, the Python routine is called with the loaded information, a new table is created, and this table is loaded to become the new LTARG table (Update path). If no detections occur for a period of time, and the vehicle is off of its intended path, it will move to get back to its path (Move Back To Path). At this point, the new LTARG table is used, and the vehicle follows the new path to avoid the obstacle. Time advances for 3 seconds and then it moves to the next event, awaiting a new detection (Returned To Path). Upon encountering a new obstacle, the event advances again, repeating the process (New Detect). Again, if no obstacles are seen for a time, and the vehicle is not on its intended path, it will move to get back on its path (Move Back To Path). After executing a move to move back to path, the event will advance to again wait for new detections (Returned To Path).

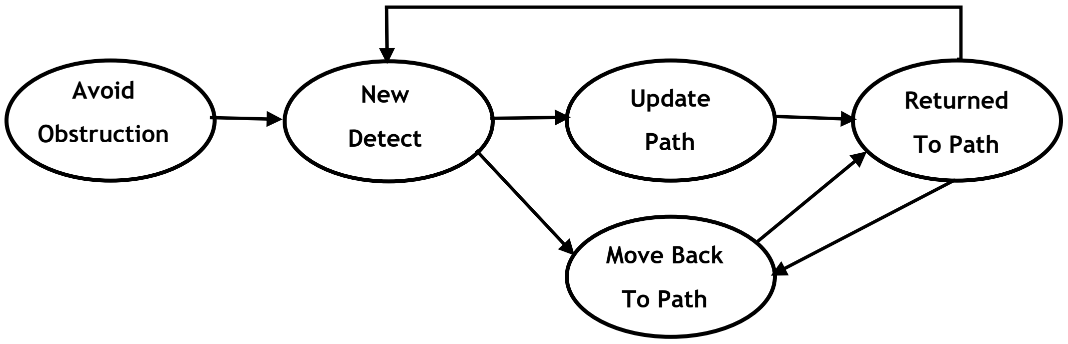


Figure 15 Event sequence for the collision avoidance simulation.

Generating a new path in Python

The Python routine used to generate the new path is similar to what is used in Example 5 (“Changing a Target path”). The main difference is that the routine can accommodate multiple detections when plotting a new path. It also has an additional routine to put the vehicle back on its target path it is off the path and no objects are nearby.

A routine has been added to check for collisions. The code is shown in Listing 2. The routine uses a simple metric of checking whether the vehicle center is within 1.5 meters of the center of an obstacle. More complex checks can be performed if desired (for example, using the vehicle dimensions and yaw to determine the four corner points of the vehicle) and near misses can be noted by analyzing the distance more carefully. Using Python, the variable strings are created for each object which gives the routine access to their locations. Since the objects could be moving, the locations are updated with each call. For this particular simulation, it also rejects objects with zero height (these are actually the preview points) and eliminates them from consideration for collision.

If the user is concerned that collision detection may be too time-consuming, the FMOD( ) operator can be used to only check for collisions once every N time steps. In this example, the collision detection routine is run once every 200 time steps, or about every .2 seconds of simulation time.

Note When the Embedded Python accesses a variable or output directly (‘vs.getval(“Bearing”)’) it is important to know that the value retrieved are in the variables base units for the unit type accessed. For example, an variable “Bearing” might have units of degrees, and if used an output, will present the angle in units. But the core value (and the value retrieved by the Embedded Python) will be in radians. Be careful when accessing variables with units that are not the base units (meters, radians, seconds, etc.).

Listing 3. *Python routine used to detect collisions based on vehicle and objects locations.*

```python
import math
import vs

def collisions (signal, intab):

    num_objects = int(vs.getval("NUM_OBS"))
    #Look through all objects, to see if they are close to current
    #position of vehicle.
    vs.debug(0)

    #Get position of vehicle
   lat_veh = vs.getval("Lat_Veh")
   stat_veh = vs.getval("Station")
   veh_x = vs.path_x_id(stat_veh,lat_veh,1,1)
   veh_y = vs.path_y_id(stat_veh,lat_veh,1,1)
   #vs.print(str(num_objects))
   #vs.print(str(veh_x))
   #vs.print(str(veh_y))

   for obj in range(1, (num_objects+1)):
      sobj = str(int(obj))
      object_ht = vs.getval("H_OBJ("+sobj+")")

      # Avoid 0 height objects
      if (object_ht > 0.0):
         obj_lat = vs.getval("LatO_"+sobj)
         obj_stat = vs.getval("S_Obj_"+sobj)
         obj_x = vs.path_x_id(obj_stat,obj_lat,1,1)
         obj_y = vs.path_y_id(obj_stat,obj_lat,1,1)
         dist = (veh_x-obj_x)*(veh_x-obj_x)+(veh_y-obj_y)*(veh_y-obj_y)
         dist = math.sqrt(dist)
         if (dist < 1.5):
             collision_time = vs.getval("COLLISION_TIME")
             if (collision_time <= 0.0):
                 vs.var("COLLISION_TIME").setvalue(vs.getval("T"))
             #vs.print(str(dist))
             break

   return 0.0
```

If the position of the last deer is adjusted as mentioned above, then the vehicle will collide with the deer. The collision will be detected and noted on the simulation display. A screen shot showing the point of collision is shown in Figure 16.

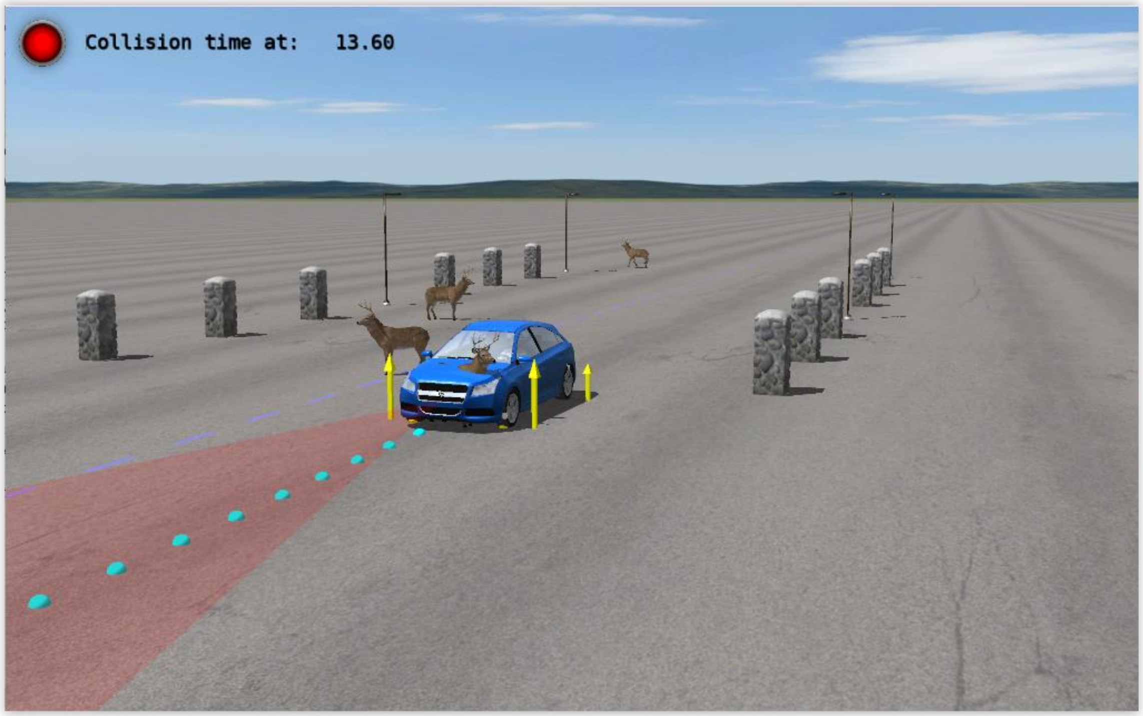


Figure 16 Simulation with modified obstacle position resulting in a collision.

Example 7: Optimizing a Simulation Parameter

This section describes another example which uses the Embedded Python to investigate and optimize a simulation parameter. In this example, a simple steering model is optimized with respect to its L_FORWARD parameter (This is the parameter that determines the distance the preview point is ahead of the vehicle.) by assessing how closely the vehicle path of the simulation matches the desired vehicle path. Ideally, this example can function as a template for users who wish to optimize a parameter, based on a metric that can be measured.

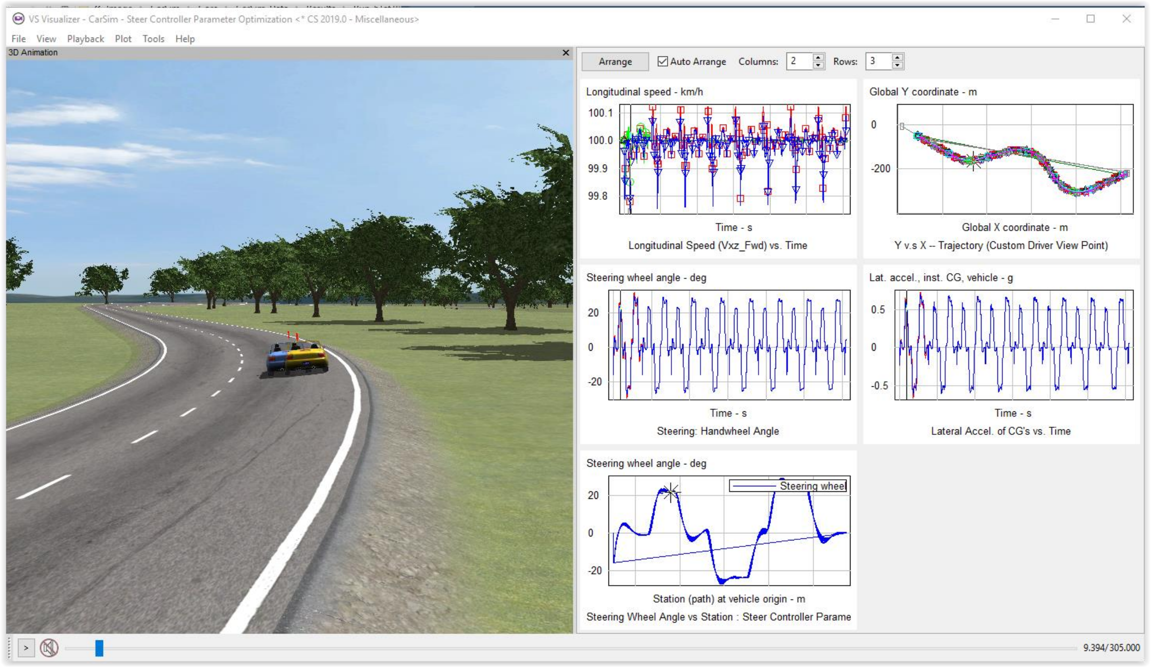

VS Visualizer - CarSim - Steer Controller Parameter Optimization <* CS 2019.0 - Miscellaneous>
File View Playback Plot Tools Help
3D Animation
Arrange
Auto Arrange
Columns: 2
Rows: 3
Longitudinal speed - km/h
Time - s
Longitudinal Speed (Vxz_Fwd) vs. Time
Global Y coordinate - m
Global X coordinate - m
Y v.s X -- Trajectory (Custom Driver View Point)
Steering wheel angle - deg
Time - s
Steering: Handwheel Angle
Lat. accel., inst. CG, vehicle - g
Time - s
Lateral Accel. of CG's vs. Time
Steering wheel angle - deg
Steering wheel
Station (path) at vehicle origin - m
Steering Wheel Angle vs Station : Steer Controller Parame
9.394/305.000

Figure 17. Simulation of vehicle that optimizes a parameter.

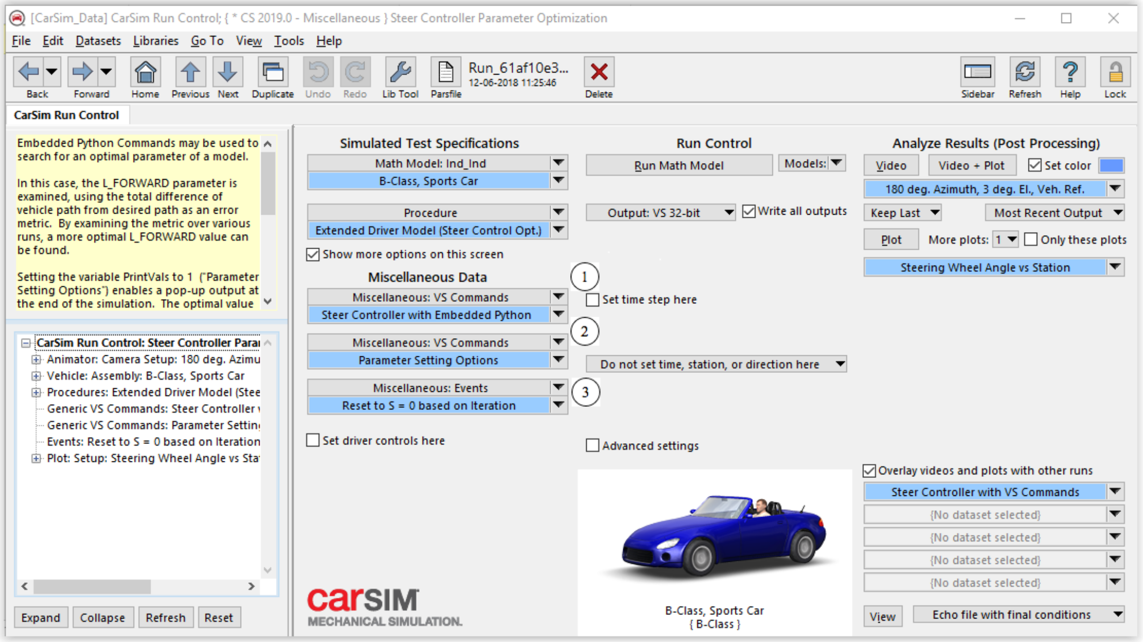
Overview of the Simulation

A vehicle is directed to follow a path on a long slowly curving road. This is the same path used to demonstrate the Embedded Python Steering Controller. The simulation also uses the simple Steering Controller for optimization. The path followed is designated with the parameter `LAT_TRACK`, which is merely a constant offset -1.6 meters from the road center. The simulation endeavors to have the vehicle follow this path. The simulation uses events to travel the road several times. For each iteration a different value of L_FORWARD is used for the Steering Model. A metric is developed to measure how much the vehicle deviates from the desired path with each iteration. This metric is used to evaluate how well the L_FORWARD value worked for the simulation. In this case, the lower the metric value, the better performance of the simulation (least deviation from the desired path). After all the iterations have been run, the best value of L_FORWARD (the value that produces the smallest deviation from the desired path) is recorded and can be noted or output.

Implementation

This example uses Embedded Python to calculate the metric and record the best values encountered so far. A simple table is used to hold the values of L_FORWARD to test, but more sophisticated strategies could also be employed. This simulation only runs a fixed number of iterations, but again, a more sophisticated strategy could also be employed to have a variable number of iterations, if desired. The user also has the option for the output results to be displayed with a Pop-Up screen (the simulation ends). To activate this screen, a single variable, PrintVals needs to be changed from 0 to 1.

The run screen for the Parameter Optimizing Simulation is shown below (Figure 18).


[CarSim_Data] CarSim Run Control; { * CS 2019.0 - Miscellaneous } Steer Controller Parameter Optimization

Run_61af10e3...
12-06-2018 11:25:46

Embedded Python Commands may be used to search for an optimal parameter of a model.

In this case, the L_FORWARD parameter is examined, using the total difference of vehicle path from desired path as an error metric. By examining the metric over various runs, a more optimal L_FORWARD value can be found.

Setting the variable PrintVals to 1 ("Parameter Setting Options") enables a pop-up output at the end of the simulation. The optimal value

CarSim Run Control: Steer Controller Para
Animator: Camera Setup: 180 deg. Azimu
Vehicle: Assembly: B-Class, Sports Car
Procedures: Extended Driver Model (Stee
Generic VS Commands: Steer Controller
Generic VS Commands: Parameter Settin
Events: Reset to S = 0 based on Iteration
Plot: Setup: Steering Wheel Angle vs Sta

Simulated Test Specifications
Math Model: Ind_Ind
B-Class, Sports Car
Procedure
Extended Driver Model (Steer Control Opt.)
[✓] Show more options on this screen
Miscellaneous Data
Miscellaneous: VS Commands
Steer Controller with Embedded Python
Miscellaneous: VS Commands
Parameter Setting Options
Miscellaneous: Events
Reset to S = 0 based on Iteration
[ ] Set driver controls here

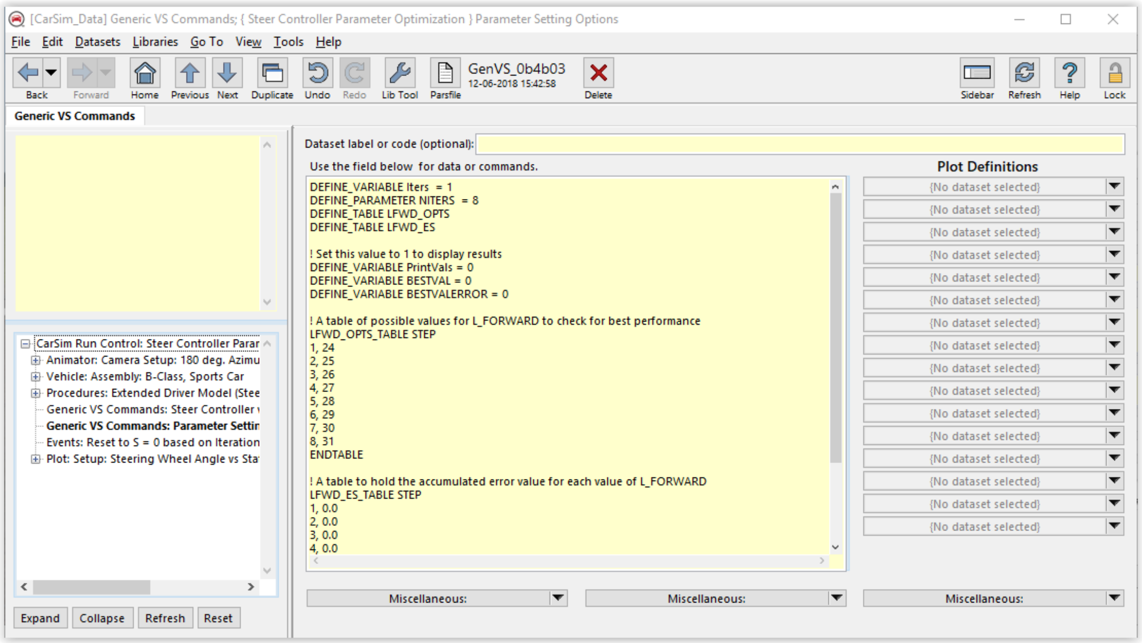
Run Control
Run Math Model
Output: VS 32-bit
[✓] Write all outputs
[ ] Set time step here
Do not set time, station, or direction here
[ ] Advanced settings

Analyze Results (Post Processing)
[✓] Set color
180 deg. Azimuth, 3 deg. El., Veh. Ref.
Keep Last
Most Recent Output
More plots: 1
[ ] Only these plots
Steering Wheel Angle vs Station
[✓] Overlay videos and plots with other runs
Steer Controller with VS Commands
{No dataset selected}
{No dataset selected}
{No dataset selected}
{No dataset selected}
Echo file with final conditions

B-Class, Sports Car
{ B-Class }

Figure 18. Run Screen for parameter optimization simulation using Embedded Python.

The run screen has three links of particular note, (1) the link to the commands that define the Steer Controller with Embedded Python (this is similar to the simple Steering Controller example used in other simulations), (2) the link that contains the tables and code to update the parameter for each iteration, (3) the link to the events sequence which iterates the simulation through a number of attempts, and records the results. The screen showing the parameter update is shown below (Figure 19).

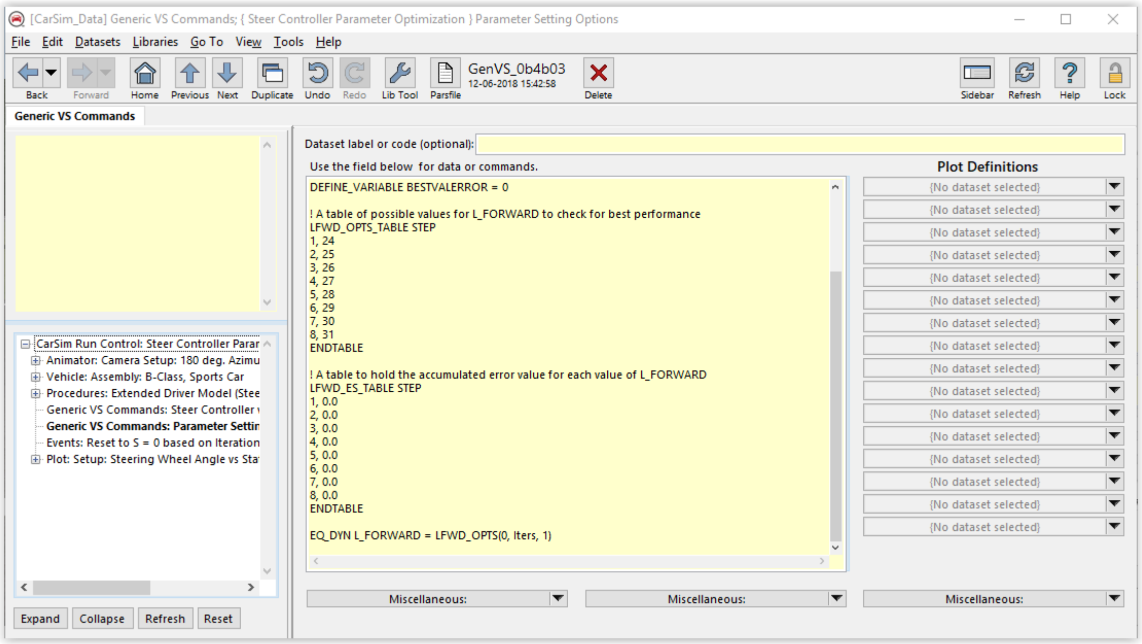

[CarSim_Data] Generic VS Commands; { Steer Controller Parameter Optimization } Parameter Setting Options
File Edit Datasets Libraries Go To View Tools Help
Back Forward Home Previous Next Duplicate Undo Redo Lib Tool Parsfile
GenVS_0b4b03
12-06-2018 15:42:58
Delete Sidebar Refresh Help Lock
Generic VS Commands
CarSim Run Control: Steer Controller Parar
Animator: Camera Setup: 180 deg. Azimu
Vehicle: Assembly: B-Class, Sports Car
Procedures: Extended Driver Model (Stee
Generic VS Commands: Steer Controller
Generic VS Commands: Parameter Settin
Events: Reset to S = 0 based on Iteration
Plot: Setup: Steering Wheel Angle vs Sta
Expand Collapse Refresh Reset
Dataset label or code (optional):
Use the field below for data or commands.
Plot Definitions
{No dataset selected}
Miscellaneous:

```
DEFINE_VARIABLE Iters  = 1
DEFINE_PARAMETER NITERS  = 8
DEFINE_TABLE LFWD_OPTS
DEFINE_TABLE LFWD_ES

! Set this value to 1 to display results
DEFINE_VARIABLE PrintVals = 0
DEFINE_VARIABLE BESTVAL = 0
DEFINE_VARIABLE BESTVALERROR = 0

! A table of possible values for L_FORWARD to check for best performance
LFWD_OPTS_TABLE STEP
1, 24
2, 25
3, 26
4, 27
5, 28
6, 29
7, 30
8, 31
ENDTABLE

! A table to hold the accumulated error value for each value of L_FORWARD
LFWD_ES_TABLE STEP
1, 0.0
2, 0.0
3, 0.0
4, 0.0
```

Figure 19. Parameter Update Screen for Parameter Optimizing Simulation (upper).

What is notable here are several variables defined to support checking different parameter choices. One table holds the different values to be used for the L_FORWARD parameter. Another table is defined to hold the metric result that indicates the suitability of a particular parameter for the simulation vehicle to closely follow the desired path. These metric results for a parameter are compiled during each iteration using that parameter. The metric in this case is just a sum of the absolute value of the lateral error (distance of the vehicle's position from the desired path) that is calculated each time the steering value is updated.

Another variable of interest is the PrintVals variable. If set to one, then a Pop-Up screen will occur at the end of the simulation to show the metric results for all of the parameter choices and the best value for the parameter, based on the metric. If not using the Pop-Up screen (nominal operation) then the user can see the best value of L_FORWARD by viewing the 'Echo file with final conditions' output file and noting the value of 'BESTVAL'. The metric value for this setting is found in 'BESTVALERROR'. An ending screen showing the Pop-Up screen with the simulation results is shown in Figure 21.

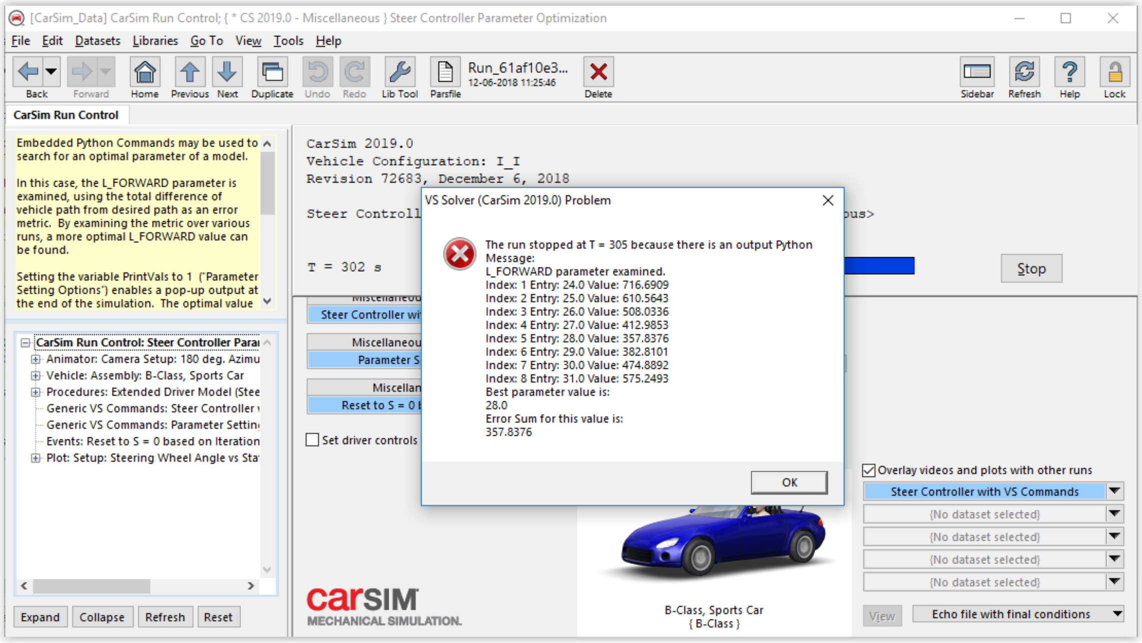

[CarSim_Data] Generic VS Commands; { Steer Controller Parameter Optimization } Parameter Setting Options
File Edit Datasets Libraries Go To View Tools Help
Back Forward Home Previous Next Duplicate Undo Redo Lib Tool Parsfile
GenVS_0b4b03
12-06-2018 15:42:58
Delete Sidebar Refresh Help Lock
Generic VS Commands
CarSim Run Control: Steer Controller Parar
Animator: Camera Setup: 180 deg. Azimu
Vehicle: Assembly: B-Class, Sports Car
Procedures: Extended Driver Model (Stee
Generic VS Commands: Steer Controller
Generic VS Commands: Parameter Settin
Events: Reset to S = 0 based on Iteration
Plot: Setup: Steering Wheel Angle vs Sta
Expand Collapse Refresh Reset
Dataset label or code (optional):
Use the field below for data or commands.
Plot Definitions
{No dataset selected}
Miscellaneous:
Miscellaneous:
Miscellaneous:

```
DEFINE_VARIABLE BESTVALERROR = 0

! A table of possible values for L_FORWARD to check for best performance
LFWD_OPTS_TABLE STEP
1, 24
2, 25
3, 26
4, 27
5, 28
6, 29
7, 30
8, 31
ENDTABLE

! A table to hold the accumulated error value for each value of L_FORWARD
LFWD_ES_TABLE STEP
1, 0.0
2, 0.0
3, 0.0
4, 0.0
5, 0.0
6, 0.0
7, 0.0
8, 0.0
ENDTABLE

EQ_DYN L_FORWARD = LFWD_OPTS(0, Iters, 1)
```

Figure 20. Parameter Update Screen for Parameter Optimizing Simulation (lower).

This bottom portion of the screen is shown merely to show how the L_FORWARD parameter is updated. Using an EQ_DYN, the value is assigned from the values table, based on the iteration being run. The iteration variable, ITERS, is updated via the events sequence.

The events sequence increases the iteration count if the vehicle has reached a station near the end of the road. It then stores the calculate error metric and resets the vehicle to Station 0, updates the L_FORWARD parameter to a new value, and restarts the error accumulation. After the desired number of iterations have been run, the simulation ends.


[CarSim_Data] CarSim Run Control; { * CS 2019.0 - Miscellaneous } Steer Controller Parameter Optimization
File Edit Datasets Libraries Go To View Tools Help
Back Forward Home Previous Next Duplicate Undo Redo Lib Tool Parsfile
Run_61af10e3...
12-06-2018 11:25:46
Delete Sidebar Refresh Help Lock
CarSim Run Control
CarSim Run Control: Steer Controller Parai
Animator: Camera Setup: 180 deg. Azimu
Vehicle: Assembly: B-Class, Sports Car
Procedures: Extended Driver Model (Stee
Generic VS Commands: Steer Controller
Generic VS Commands: Parameter Settin
Events: Reset to S = 0 based on Iteration
Plot: Setup: Steering Wheel Angle vs Sta
Expand Collapse Refresh Reset
Stop
Steer Controller wi
Miscellaneou
Parameter S
Miscellan
Reset to S = 0 b
Set driver controls
Overlay videos and plots with other runs
Steer Controller with VS Commands
{No dataset selected}
View
Echo file with final conditions
B-Class, Sports Car
{ B-Class }
carSIM
MECHANICAL SIMULATION.
OK

Embedded Python Commands may be used to search for an optimal parameter of a model.

In this case, the L_FORWARD parameter is examined, using the total difference of vehicle path from desired path as an error metric. By examining the metric over various runs, a more optimal L_FORWARD value can be found.

Setting the variable PrintVals to 1 ("Parameter Setting Options") enables a pop-up output at the end of the simulation. The optimal value

```
CarSim 2019.0
Vehicle Configuration: I_I
Revision 72683, December 6, 2018

Steer Controll

T = 302 s
```

VS Solver (CarSim 2019.0) Problem

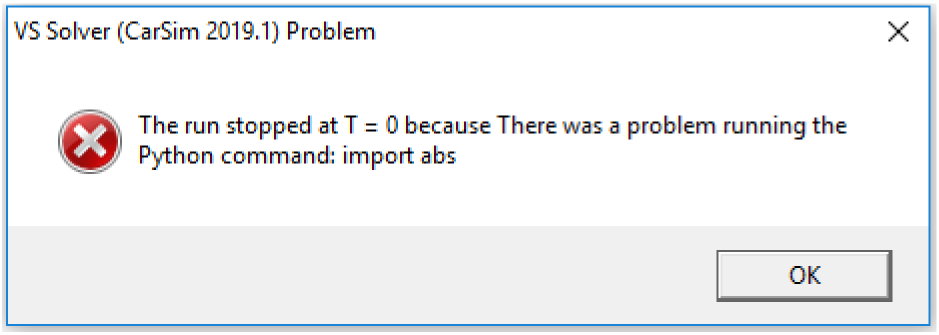
```
The run stopped at T = 305 because there is an output Python
Message:
L_FORWARD parameter examined.
Index: 1 Entry: 24.0 Value: 716.6909
Index: 2 Entry: 25.0 Value: 610.5643
Index: 3 Entry: 26.0 Value: 508.0336
Index: 4 Entry: 27.0 Value: 412.9853
Index: 5 Entry: 28.0 Value: 357.8376
Index: 6 Entry: 29.0 Value: 382.8101
Index: 7 Entry: 30.0 Value: 474.8892
Index: 8 Entry: 31.0 Value: 575.2493
Best parameter value is:
28.0
Error Sum for this value is:
357.8376
```

Figure 21. End Pop-Up Screen for Parameter Optimizing Simulation when PrintVals = 1.

The simulation demonstrates, in a very simple and straightforward way, how a simulation can be iterated through different values of a single parameter. A metric is calculated to evaluate the relative worthiness of a parameter choice compared with the other choices that are considered. Based on the metric, an optimized parameter can be selected.

Debugging Embedded Python Code

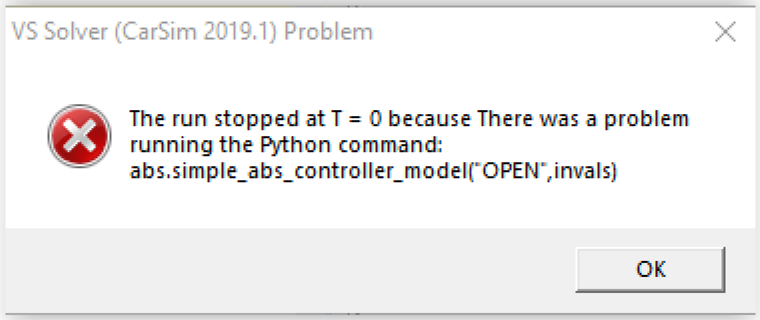
Unfortunately, there is not a perfect solution to debug an embedded Python code. When the Run Math Model button is pressed, if there is an error in Python script, it displays an error message which often does not help locate an exact line of the problem (Figure 22).



```
The run stopped at T = 0 because There was a problem running the
Python command: import abs
```

Figure 22. Embedded Python Code Error Message (Import).

Actually, this error provides a small amount of information. Because the error occurs with "import abs", it means that there was a problem loading the module abs.py. If it were a run-time error, the error message would look something like this (Figure 23):



```
The run stopped at T = 0 because There was a problem
running the Python command:
abs.simple_abs_controller_model("OPEN",invals)
```

Figure 23 Embedded Python Code Error Message (Run-time).

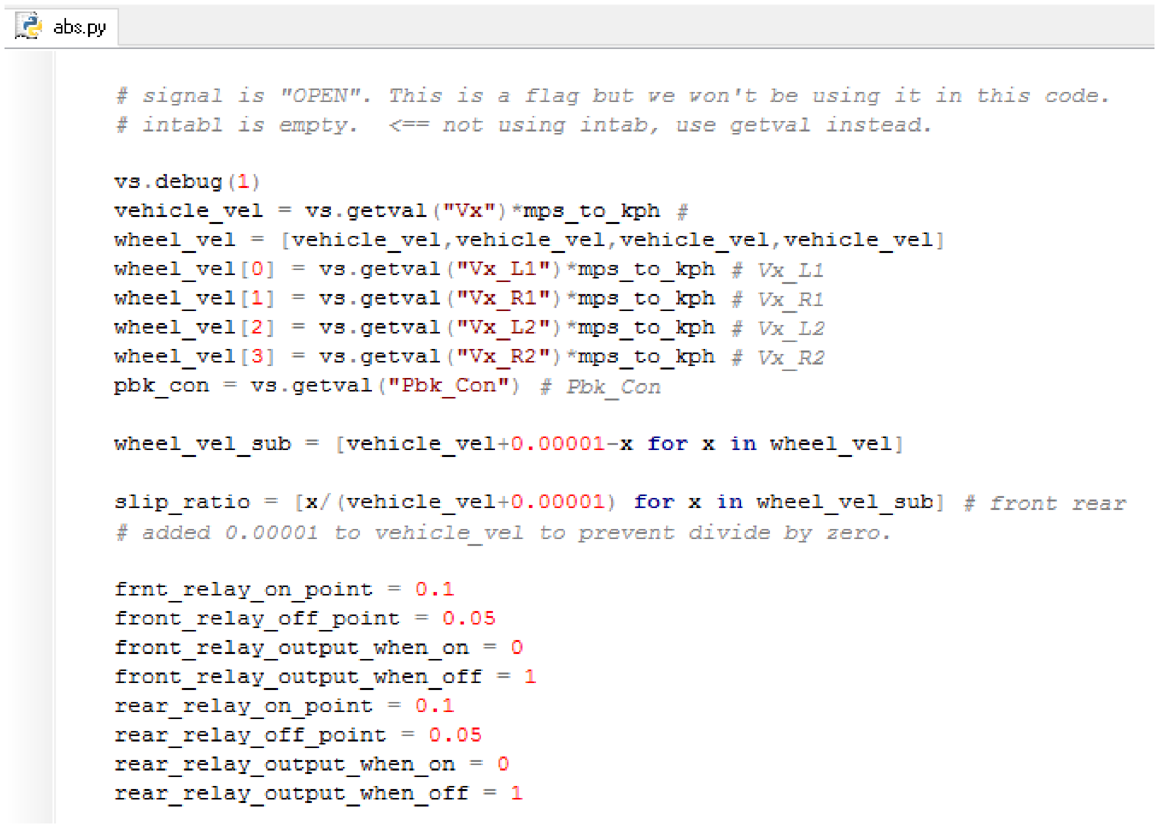
However, depending on types of errors, it is possible to narrow down the location of the problem by using `vs.debug` or `vs.print` commands.

Figure 24 shows an example of how to use `vs.debug` function. To turn the debug function on, use a value 1 as an argument. When `vs.debug` is turned on, then many calls made to the vs module are noted and printed out in a diagnostic window. This can help the user to determine which part of the code is causing a problem or error.

In Figure 24, a variable, `front_relay_on_print`, is intentionally misspelled as `frnt_relay_on_print`. When the simulation is executed, a pop-up message is displayed and informs that all getval and setval in the code up until the error has occurred. In this example, it informs that it has processed at least up to the line of code, `getval: Pbk_Con` (Figure 25), but did not reach the line of code `var: pbk_L1` (about 30 lines later), so the user can somewhat narrow down where the problem occurs. The user can also add several `vs.print()` functions in the code instead of using `vs.debug`. For example, vs.print("Pass 1") is added in earlier part of the code, and vs.print("Pass 2") can be added later part of the code. When executed, a pop-up message is displayed with these print statements.

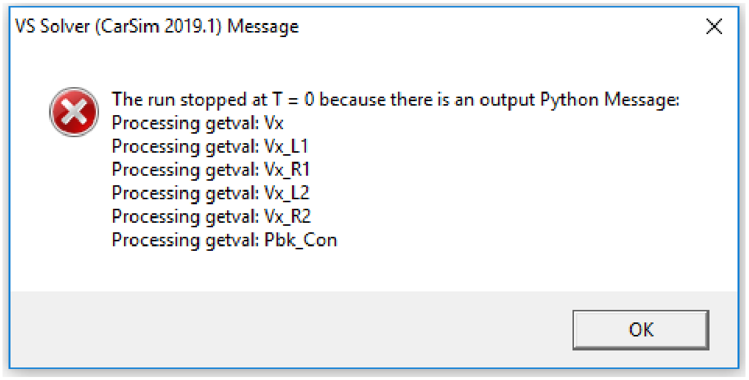
The vs.print() and vs.debug() commands can print out with each time step, so they are usually not practical to be active when running working code. To halt a simulation which is displaying a print or debug screen with each time step, simply hit the main 'stop' button on the paused simulation control, and then close the print or debug window. Sometimes, the user may have to move the print/debug window to access the simulation 'stop' button.

As mentioned above, unfortunately, `vs.debug()` and `vs.print()` can be useful for only certain types of the error. Therefore, adding and testing a small block of code at a time is recommended when writing an embedded Python program.



```
# signal is "OPEN". This is a flag but we won't be using it in this code.
# intabl is empty.  <== not using intab, use getval instead.

vs.debug(1)
vehicle_vel = vs.getval("Vx")*mps_to_kph #
wheel_vel = [vehicle_vel,vehicle_vel,vehicle_vel,vehicle_vel]
wheel_vel[0] = vs.getval("Vx_L1")*mps_to_kph # Vx_L1
wheel_vel[1] = vs.getval("Vx_R1")*mps_to_kph # Vx_R1
wheel_vel[2] = vs.getval("Vx_L2")*mps_to_kph # Vx_L2
wheel_vel[3] = vs.getval("Vx_R2")*mps_to_kph # Vx_R2
pbk_con = vs.getval("Pbk_Con") # Pbk_Con

wheel_vel_sub = [vehicle_vel+0.00001-x for x in wheel_vel]

slip_ratio = [x/(vehicle_vel+0.00001) for x in wheel_vel_sub] # front rear
# added 0.00001 to vehicle_vel to prevent divide by zero.

frnt_relay_on_point = 0.1
front_relay_off_point = 0.05
front_relay_output_when_on = 0
front_relay_output_when_off = 1
rear_relay_on_point = 0.1
rear_relay_off_point = 0.05
rear_relay_output_when_on = 0
rear_relay_output_when_off = 1
```

Figure 24. Using vs.debug().



VS Solver (CarSim 2019.1) Message

```
The run stopped at T = 0 because there is an output Python Message:
Processing getval: Vx
Processing getval: Vx_L1
Processing getval: Vx_R1
Processing getval: Vx_L2
Processing getval: Vx_R2
Processing getval: Pbk_Con
```

Figure 25. Message When vs.debug(1) is Used.

Note The vs.table object is useful for getting values, (sometimes) setting values, and interrogating the contents of VS tables:

```
vs.tab(“TAB_NAME”).num_rows()
vs.tab(“TAB_NAME”).num_cols()
```

Find the extent of 1-D (and 2-D) tables.

```
vs.tab(“TAB_NAME”).entry(c,r,n).value()
vs.tab(“TAB_NAME”).entry(c,r,n).set_value(value)
```

Get (or sometimes Set) a Table value based on x-value (*r*), y-value (*c*) and table index (*n*) if there are multiple tables in a group.

```
vs.tab(“TAB_NAME”).entry(i,j,n).defvalue()
vs.tab(“TAB_NAME”).entry(-1,j,n).defvalue()
vs.tab(“TAB_NAME”).entry(i,-1,n).defvalue()
```

Get the defined values (how the table was initially defined) of a table where *i* is the column index, *j* is the row index, and *n* is the table index. If you set *i* or *j* to -1, you can retrieve the x or y entry values, respectively.

Installing Python packages

Numerous packages are available to expand the capabilities of Python. If an Embedded Python application makes use of a non-standard package or library (such as Pandas or NumPy) then the Python needs to have the package installed. This is usually done with Python’s package manager, **pip**. If an outside Python distribution is being used, then that can be updated in the standard manner. If one of the internal Python distributions (32-bit and 64-bit) is used, then that distribution also needs to be updated with the new package or packages.

Note If an outside Python distribution has been updated, it is possible that the solver may not recognize all of the updates. The user may need to add one or more directory paths for the simulation to work. This can be done using the Python command `sys.path.append()`. In particular, the '`site-packages`' sub-directory in the Python distribution area may need to be added.

To update an internal Python distribution on Windows, one must do the following:

1. One needs an outside Python 3.X installed on the machine that is that same word size (32 or 64 bits) as the internal distribution to be updated. It should match the distribution version of the internal Python as close as possible.
2. Start up a command screen or powershell screen.
3. Verify that the default Python and **pip** refer to the Python distribution mentioned in the first step. This can be done with 'where python' and 'where pip' typed on the command screen. You may have to change the default Python if necessary, to match the internal Python distribution to be updated.
4. Change directory so that the command screen is at …\Programs\Python\Python32. (Or ...\Python64 if installing to the 64-bit Python)
5. To install, for example, "NumPy", type the following on the command line: **pip install numpy --python-version 3.10 --no-deps -t .** (The version should be version of the internal python distribution, which as of 2022.1 is 3.10.2) [Some packages do have dependencies, in which case you should leave off the **--no-deps** flag, and follow any subsequent prompts.]
6. This should install the package. For Linux systems, the same analogous steps apply. Under Linux, the 'which' command is used to determine which Python and **pip** are being used.

Note Some packages (notably NumPy) do not close properly when run from embedded Python distributions. This is a problem or bug with NumPy in that it does not close properly unless the parent process is ended. The behavior is that a simulation using NumPy can run once, but if the browser interface is run again, the embedded Python fails. This can be problematic if using the browser to run and develop a simulation.

There is a simple workaround, however. If one calls the solver using an external wrapper, then the process is not associated with the browser process, and the problem does not occur. This can be done by selecting 'Models: Self-Contained Solvers' on the run screen, and then selecting either the 32-bit or 64-bit External Wrapper programs (included) to run the solver. The Braking example uses a solver wrapper to show how this is done.

Summary

This document shows how a VS Math Model can be extended using the Embedded Python utility. Three examples are provided; one is a nominal steering controller, which has been implemented by other means (including ordinary VS commands) previously; another example creates new target paths as needed to avoid novel obstacles a vehicle may encounter in a simulation; and the third example runs multiple iterations to find an optimal value to a simulation parameter.

The examples are intended to show the breadth of capability that is feasible with the Embedded Python utility.

Basically, by making use of the Embedded Python utility to extend the model:

1. No additional software needed beyond CarSim/TruckSim/BikeSim. Some type of utility or package is needed to create and edit Python routines.
2. Variables and parameters can be sent to the Python utility via the input table. There is no limit to the number of input values that can be sent to the utility.
3. Based on inputs provided with the input table, the user has a complete, advanced programming language (Python) to develop a new algorithm or routine which can process the input data and produce output(s) for use by the simulation.
4. There is no limit to the number of outputs (real numbers) that can be sent from the utility back to the simulation.
5. The basic commands needed to access the utility are `RUN_PYTHON_STRING` and `RUN_PYTHON_PROG`.

Other methods are available that use the VS API to run a VS Solver from the control of another program. These external programs can be written in MATLAB, C, and even Python. The Embedded Python utility serves a different purpose; control need not be given to an external program (though that is allowed) but an internal routine needed to update a simulation variable can now be written in a full-featured language to complement the existing VS Commands. Moreover, since Python is interpretive, the routine can be modified and updated without going through a compilation step or employing 3rd party packages. The user thus has a new tool to aid in low level routine development for their simulations.