755 Phoenix Drive, Ann Arbor MI, 48108, USA
Phone: 734 668-2930 • Fax: 734 668-2877 • Email: info@carsim.com
carsim.com

Extending VS Math Models with VS Commands and the VS API

This technical memo shows by example how a VS Math Model is extended to include an alternative path following controller. In this example, the controller is used to provide a simple addition to the model; it is not intended to replace the built-in path follower.

The vehicle VS Solver libraries for CarSim, TruckSim, and BikeSim are used to build VS Math Models that predict the behavior of conventional vehicles in response to driver/rider controls.

A VS Math Model can be extended to handle custom controllers and alternative component models using an external simulation environment such as Simulink or LabVIEW. There are at least three additional approaches for extending a VS Math Model:

1. Equations can be added at runtime using VS Commands.
2. Variables can be exchanged with an external custom program written in any programming language that can load a Windows dynamically linked library (DLL) file and access its functions. This is the same technique as used with Simulink and LabVIEW, applied with custom user-defined tools.
3. Variables in the VS Math Model can be accessed directly by a custom program written in C/C++. Calculations in the custom program can be tightly integrated with the calculations built into the VS Math Model.

This memo describes examples to demonstrate the above three methods for extending VS Math Models in CarSim, TruckSim, and BikeSim.

VS Commands are described in the *VS Commands Reference Manual* (VS Commands) and require no additional software beyond a VehicleSim product from Mechanical Simulation such as CarSim. The other methods are described in *The VehicleSim API: Accessing and Extending VS Solver Programs*. The example programs and datasets from this memo are included in CarSim, TruckSim, and BikeSim.

An Example Model Extension

A path preview control is used as an example model extension. The controller is based on a "preview point" in front of the vehicle and the relationship of that preview point to a target path relative to the road reference line. 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 1 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 left of the target line, and therefore the vehicle steering wheel should be turned to the right. In the case of BikeSim, where control is made by leaning the bike, the bike should be leaned to the right.

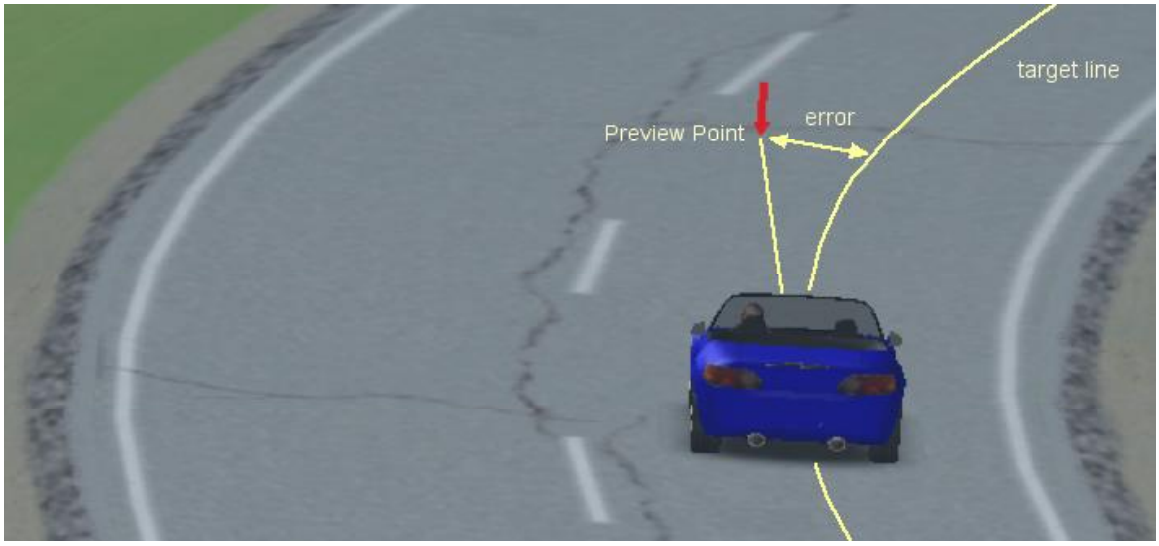


Figure 1. Vehicle with a preview point for a simple steering controller.

Note	The example controller presented in this memo is simple to describe and easy to visualize, but does not make use of some of the built-in capabilities of CarSim, TruckSim, and BikeSim such as driver preview sensors. It is provided solely to illustrate the use of VS Commands and the VS API.

Existing Model Documentation

The full list of available Import variables for a VS Math Model is generated by using the **View** button in the lower-right corner of the **Run Control** screen. 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. In BikeSim, you can import steering torque and target lean angle.

The default settings for all of the Import variables are that they be ignored. Although the native equations in the VS Math Model include the effect of the imported variables, they do not set values for the import variables. (After all, the whole purpose of importing a variable is to replace or modify the value already available in the native equations of the model.)

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 for CarSim and TruckSim; for BikeSim, the coordinates of the rider reference point are used.

Following a Path using a Preview Point

The example controllers will attempt to follow a path based on difference between a target lateral distance from a path (specified with a new parameter `LAT_TRACK`), and the lateral distance from a preview point to the path, where the point is a specified distance in front a point of interest on the vehicle. For CarSim and TruckSim, the coordinates of the preview point are:

```
Xpreview = Xcg_TM + L_FORWARD*COS(Yaw)          (1)

Ypreview = Ycg_TM + L_FORWARD*SIN(Yaw)          (2)
```

where

- `Xpreview` and `Ypreview` are X and Y coordinates of a point in front of the vehicle (new output variables),
- `Xcg_TM` and `Ycg_TM` are the names of existing output variables that provide the X and Y coordinates of the center of mass of the entire vehicle, as listed in a documentation file obtained using the **View** button on the **Run Control** screen,
- `Yaw` is the output variable for the yaw angle for the vehicle, as listed in the same documentation file, and
- `L_FORWARD` is the distance the preview point lies in front of the vehicle mass center (a new parameter).

The same equations are used for BikeSim, except the coordinates `Xcg_TM` and `Ycg_TM` are replaced with `X_Rdr` and `Y_Rdr` (global coordinates of the rider upper body) .

Steer Controller in CarSim and TruckSim

Directional control in CarSim and TruckSim is provided by either steering wheel torque or steering wheel angle, depending on the selected mode. The example controllers use steering wheel angle.

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                                    (3)
```

The example extensions will calculate the imported steering wheel angle, based on the position of the target point, which in turn depends on the current vehicle position and heading angle.

The desired steer angle during the run will be set using the import:

```
IMP_STEER_CTRL = GAIN_PATH_CTRL*
      (LAT_TRACK -ROAD_L_ID(Xpreview, Ypreview, CURRENT_ROAD_ID, 1))   (4)
```

where

`LAT_TRACK` is the target lateral position relative to the road reference path (a new parameter),

`ROAD_L_ID` is a function available in VS Math Models (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) for a specified Road ID,

`CURENT_ROAD_ID` is the ID of the current road (a parameter), and

`GAIN_PATH_CTRL` is a new parameter used to scale the control in proportion to tracking error.

The controller calculation is potentially complicated at the start of the run when not all of the output variables have been calculated. For this simple example, the steering wheel angle will be defined as zero until the simulation time `T` is greater than the start time parameter `T_START`. Therefore, equation 4 is replaced with a more complicated expression:

```
IMP_STEER_SW = IF(T > TSTART,
      GAIN_PATH_CTRL*
      (LAT_TRACK -ROAD_L_ID(Xpreview, Ypreview, CURRENT_ROAD_ID, 1)),
      0)                                                              (5)
```

This uses the `IF` special form to set the steer to 0 unless `T > T_START`.

BikeSim Lean and Steer

BikeSim has two levels of control for following a path:

1. Lean angle provides the control of the bike direction. An internal rider model (enabled with the keyword `OPT_RIDER_MODEL`) determines a lean target `GAMMA_TARG` needed to follow a specified path.
2. A steer controller (enabled with the keyword `OPT_STEER_CONTROL`) determines a steer torque from an internal model to try to obtain the target lean angle `GAMMA_TARG`.

Almost the same path follower logic is used in BikeSim, except that the controller calculates target lean. The Import variable is named `IMP_GAMMA_TARG`. The parameter `GAIN_PATH_CTRL` for

BikeSim has the same units of deg/m. There is a difference in sign convention compared to CarSim/TruckSim; steering to the left is positive, but leaning to the left is negative. Hence, the equation for lean has a minus sign:

```
IMP_GAMMA_TARG = IF(T > TSTART,
      -GAIN_PATH_CTRL*
      (LAT_TRACK -ROAD_L_ID(Xpreview, Ypreview, CURRENT_ROAD_ID, 1)),
      0)                                                                    (6)
```

Most of the following descriptions involve CarSim. Similar examples for BikeSim are included in the databases, with the distinctions being:

1. The BikeSim controller provides target lean angle, rather than steering angle,
2. The BikeSim controller uses equation 6, rather than equation 5.
3. The internal steer controller is enabled to apply steering torque in order to try to achieve the lean angle calculated in equation 6.

Adding the Controller with VS Commands

This section describes how the controller is added to a math model using VS Commands. This section is not a reference for VS Commands (that is the *VS Commands Reference Manual*), but is introductory in nature. Readers who are familiar with VS Commands and are mainly interested in the API examples may skim this section or even skip to the next one.

VS Commands for the Example

In reviewing the equations from the preceding section, we see that they involve four existing output variables (`T`, `Xcg_TM`, `Ycg_TM`, and `Yaw`), two new variables (`Xpreview` and `Ypreview`), and three new parameters (`LAT_TRACK`, `GAIN_PATH_CTRL`, and `L_FORWARD`). The new variables `Xpreview` and `Ypreview` that are used to calculate the steering can also be helpful if available for plotting, and for animating to show the preview point (see the arrow in Figure 1).

For this example, the VS Commands are placed in a dataset in the library **Generic: VS Commands** (Figure 2), and a link is made to this dataset from the **Run Control** screen.

The three new parameters are added with the command `DEFINE_PARAMETER` (2). The new output variables are defined with the command `DEFINE_OUTPUT` (3), using equations 1 and 2.

Every named parameter and variable in a VS model has associated units. As described in VS Commands, conversions are made automatically between *user units* shown for input and output files, and *internal dynamical units* used for internal calculations. For example, steering wheel angle has user units of degrees, and internal units of radians. All input angles are automatically divided by 180/π to convert from degrees to radians. For output, the angles are automatically multiplied by 180/π to convert from radians to degrees.

Existing user units are shown in the readme files and the Echo files generated with each run.

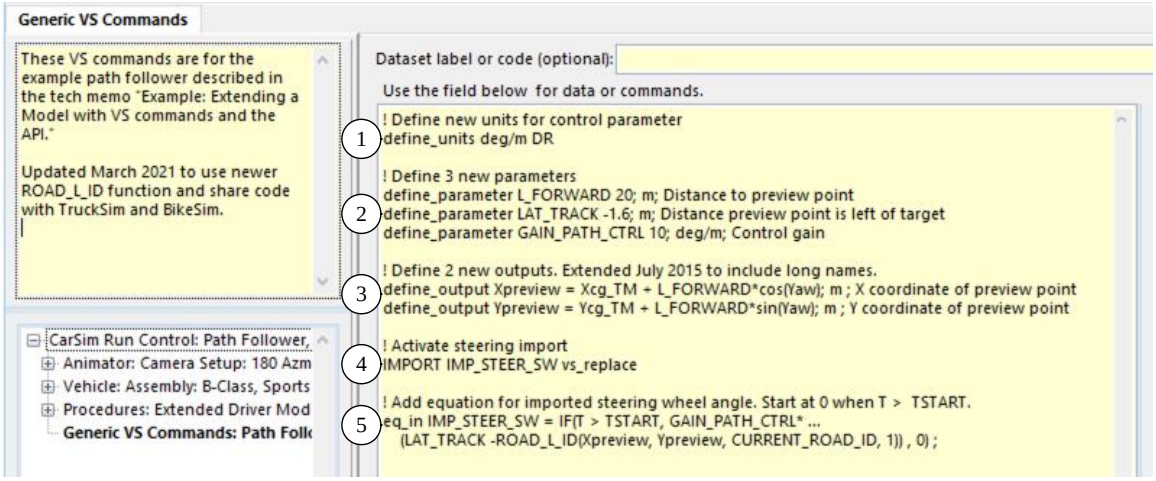


Figure 2. VS Commands for the example path follower.

> **Note** The internal units are generally SI, and by definition require no scale factors for internal calculations. The parameter `OPT_ECHO_ALL_UNITS` can be set to any nonzero value to have the VS Math Model show everything about the current user units system in the Echo file, including all of the scale factors that relate user units to internal units.

Most of the variables and parameters that will be added for this example involve length and are assigned units of meters (meters are used for most of the road geometry). The exception is the constant `GAIN_PATH_CTRL` (2), which should have units of steer or lean per unit of lateral distance: deg/m.

If a run is made with the `DEFINE_PARAMETER` command for `GAIN_PATH_CTRL`, without preparation, an error message appears saying the units "deg/m" are not installed. To allow the use of "deg/m," the `DEFINE_UNITS` command (1) is used to install everything the solver needs to know about the new units: the printed name is "deg/m" and the scale factor is set to a built-in constant `DR` (degrees/radian) = $180/\pi \approx 57.29577951$. After the new units are installed, they can be specified with the keyword `"DEG/M"` (not case sensitive), as shown in the `DEFINE_PARAMETER` command for `GAIN_PATH_CTRL` (2).

The `IMPORT` command (4) activates the imported steering wheel angle with a copy of equation 3. The last statement (5) covers two lines. The first line ends with the "…" continuation indicator. When the VS Math Model scans the line, it finds the three consecutive dots. It then clears the contents of the internal copy of the text from the location of the first dot, and continues with the first non-blank character in the next line (starting with "`(LAT_TRACK`" in this example).

At this point (after statement (5)), the controller is fully defined and the additional X and Y output coordinates are available for plotting and animation.

The VS Commands are shown in the Echo files generated by the solver programs when runs are made. Most commands appear at the end of the file (Figure 3). The printed version in the Echo file is not necessarily identical in appearance to the original commands (Figure 2), but it has the same

meaning and will give identical results if used in future runs. One difference is that all variables except outputs are shown in upper case, regardless of how they were provided as input.

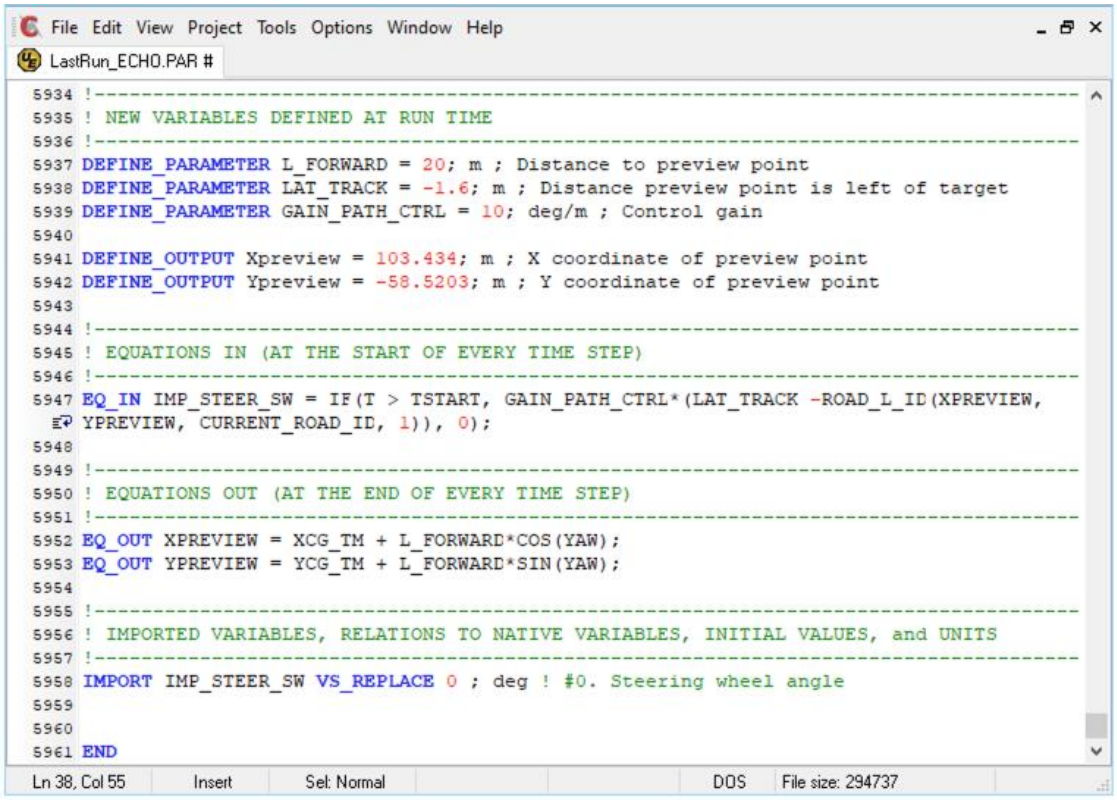


Figure 3. Most VS Commands are written at the end of an Echo file.

Note	Output variables such as `Xpreview` are represented internally with two names for a single variable. One is the keyword used to identify any reference to the variable in equations. The keyword is uppercase (e.g., `XPREVIEW`), as are the keywords for all other variables in the model. The second name, with the original mixed case (e.g., `Xpreview`) is written into the output VS or ERD file where it is available to support automatic labeling of plots.

Whenever the units system of the VS Math Model is modified (e.g., adding the new units “deg/m”), a set of commands appears at the top of the Echo file to fully specify the units system as it was modified in the run (line 16, Figure 4).

Running the Extended Model

The VS Commands can be placed anywhere in the VS database where they would be passed to the math model when it reads inputs for a new run. A simple option is to link to the VS Command dataset from the **Run Control** screen, as was done for this example.

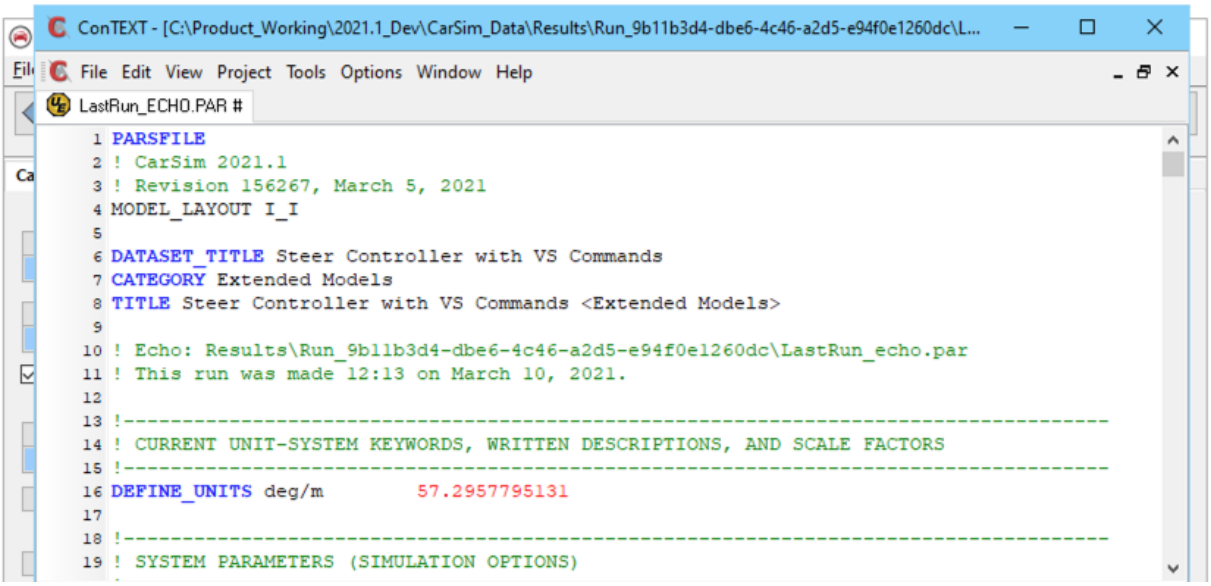


Figure 4. DEFINE_UNITS command written near the top of an Echo file.

No other steps are needed to apply the model extensions. Use the regular **Run Math Model** button to make the run, and view results with the regular **Video** and **Plot** buttons. For example, Figure 5 shows the video and two plots from an example run in CarSim. Note that the top plots shows the trajectory of several points of interest, including the preview point, identified in the plot using variable name `YPreview`. The video also shows the preview point, using a yellow arrow.

Figure 5. Results obtained with the example custom steering controller.

Extending a Model with an External Program: Overview

The tech memo *Automating Runs with the VS API* explains how to load and run a VS Math Model using a VS Solver DLL controlled by external software. This method will now be extended to include the steering controller as programmed using MATLAB and C/C++ in the following sections. In each case, a short program will load the VS Solver DLL and make the run. A program such as this that loads and runs a DLL is sometimes called a *wrapper*.

In normal operation on a Windows-based PC, the top-level input file for a VS Math Model is a simulation control file that contains pathnames for the input and output files that will be handled by the solver, plus the pathname for the VS Solver DLL. This is called the *Simfile*, and has the default name `simfile.sim`. Details of Simfile contents are provided in the reference manual for VS Math Models.

Making a run with the VS Solver involves these steps:

1. **Load**. Get a DLL pathname from a Simfile, load the VS Solver DLL, and access the VS API functions contained in the DLL.
2. **Initialize**. Read the Simfile (again) to obtain pathnames for input and output files that will be used for the run. Read input data and create and initialize a VS Math Model, whose layout is defined by input Parsfiles.
3. **Run.** Run the simulation with a simple loop in which time advances in small increments. The VS Math Model updates variables at each time step.
4. **Terminate and unload**. Terminate the run and unload the DLL.

Full details of this process are covered in the *VS API Manual*.

In the examples described in the tech memo *Automating Runs with the VS API*, the only critical VS API function used in each example wrapper program was `vs_run`, which performs steps 2 through 4 above. When a wrapper program is used to extend the model, then it is usually necessary to perform steps 2 to 4 separately. Typically, new inputs are added programmatically as part of the initialization, and additional calculations are performed each time step.

Extending the Model with MATLAB

VS Math Models are ready to run from within MATLAB/Simulink using VS S-Functions, and many users extend the models using Simulink with no knowledge or concern about the API. Alternatively, VS Math Models can be built and run directly from MATLAB (without Simulink) by using the API.

> **Note** If you do not use MATLAB and are interested in making a C wrapper, you can skip ahead to the next section (page 20).

Review: Running a VS Math Model using MATLAB

The tech memo *Automating Runs with the VS API* describes an example of running a VS Math Model from MATLAB, which will be reviewed in this subsection.

Running a VS Math Model from MATLAB is usually done with a text MATLAB M file.

When running from MATLAB, the working directory will be the folder containing the MATLAB M file. Therefore, the Simfile should be written to the folder containing the M file. This is specified using a dataset from the library **Models: Transfer to Local Windows Directory** (Figure 6). The linked dataset should identify the folder containing the MATLAB file (1). In the same dataset, use the drop-down control for choosing a 32/64 bit solver (2) to match your version of MATLAB. The examples shipped in VehicleSim products also specify a custom Simfile name, using the keyword `simfile` (3). For the examples in which no extensions are made to the VS Math Model, the Simfile is named `simple.sim`.

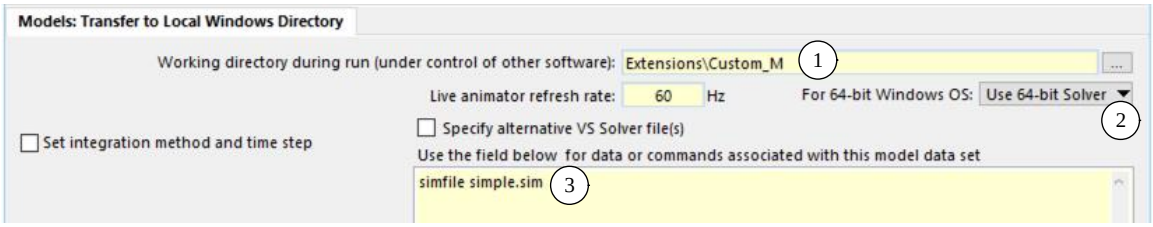


Figure 6. Specify the folder containing MATLAB files.

To use these settings, a link is made from the **Run Control** screen (Figure 7) to the dataset in the library **Models: Transfer to Local Windows Directory** (4), selected with the **Models** drop-down control (3). The linked dataset (2) is the one shown in Figure 6. Generate the files for the MATLAB program by clicking the button **Generate Files for this Run** (1).

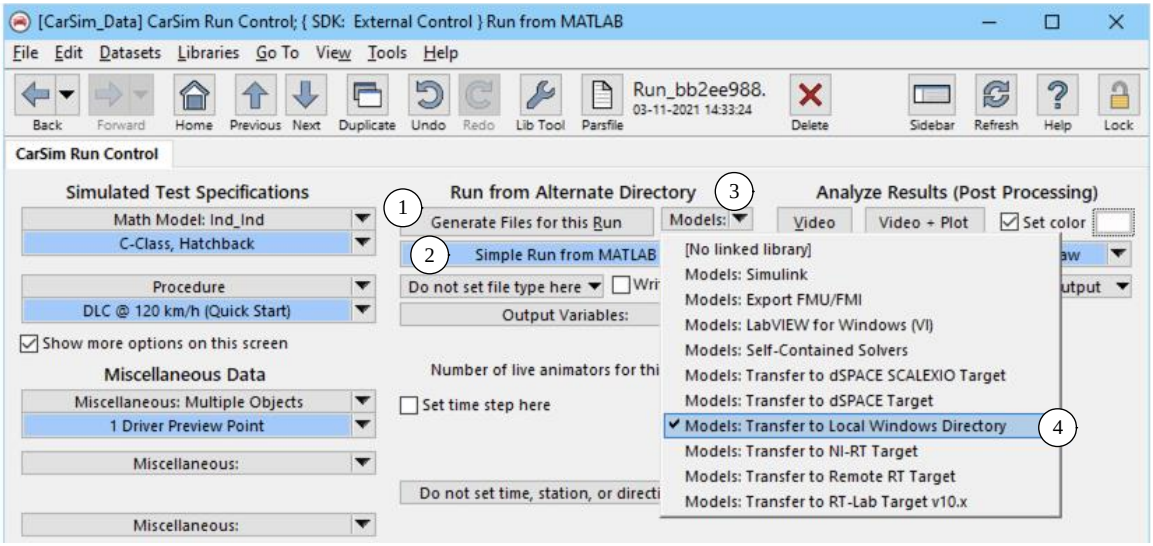


Figure 7. Link to the library Models: Transfer to Local Windows Directory.

To run the simulation from MATLAB (Figure 8), load the `simple.m` file (1) into MATLAB (2) and run it using a menu control, MATLAB command, or **Run** button (3) (the details for how you control MATLAB depend on the version). Status of the simulation is shown in the **Command Window** (4).

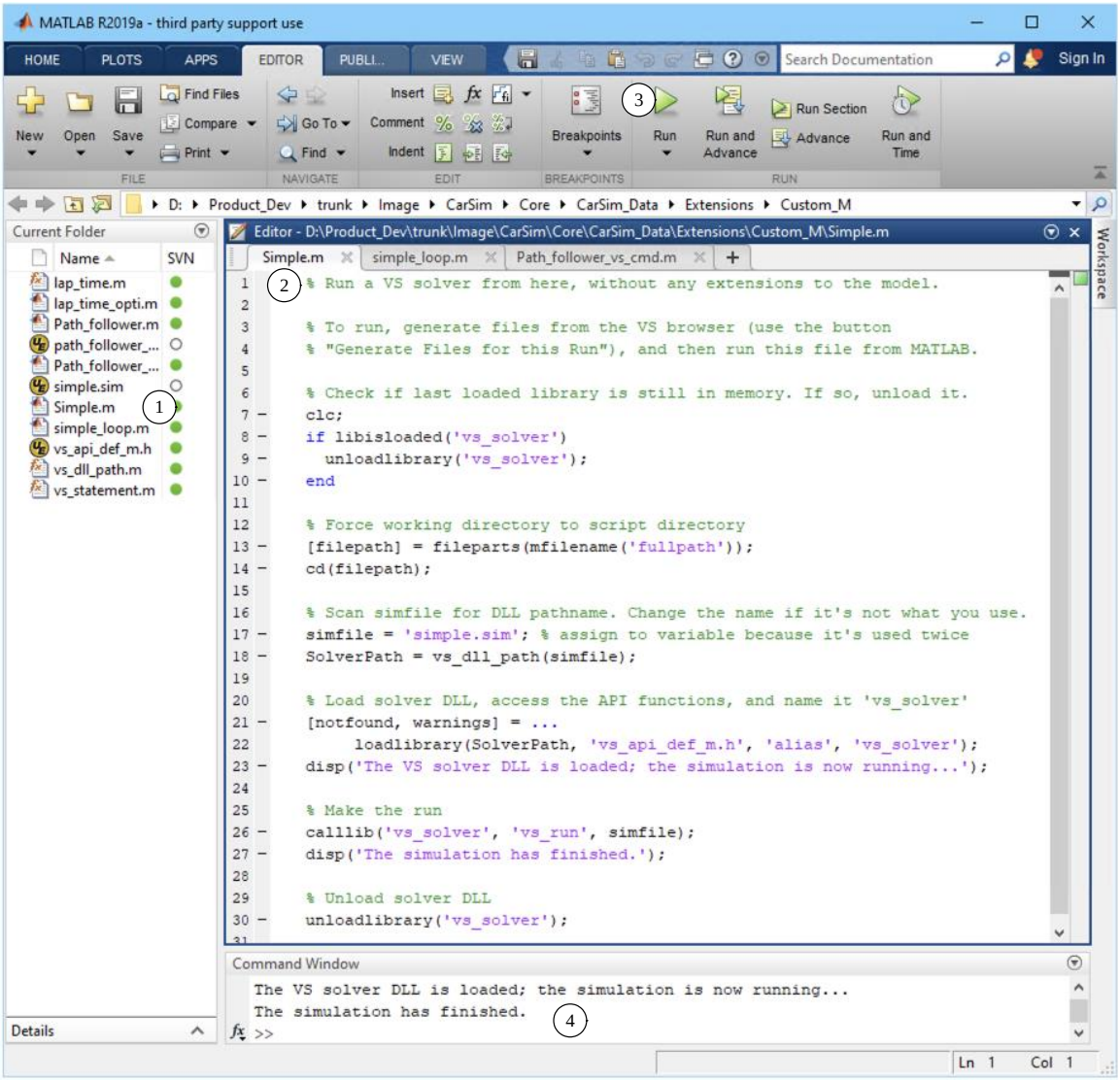


Figure 8. Running a VS Math Model using the vs_run function from MATLAB.

In this example, line 18 reads the simfile to obtain the pathname for the VS Solver DLL. Lines 21 and 22 load the solver, line 26 runs the simulation, and line 30 unloads the Solver DLL.

Running a Simulation Loop in MATLAB

Another M file `simple_loop.m` (Figure 9) performs the same purpose as `simple.m`, but uses a few more commands in order to have access to the VS Math Model at different times in the simulation.

Both of these M files have the same code at the start and finish. At the start, both have the same commands going to `loadlibrary` (lines 23 and 24, Figure 9). Both finish with the `unloadlibrary` command (line 47 in Figure 9).

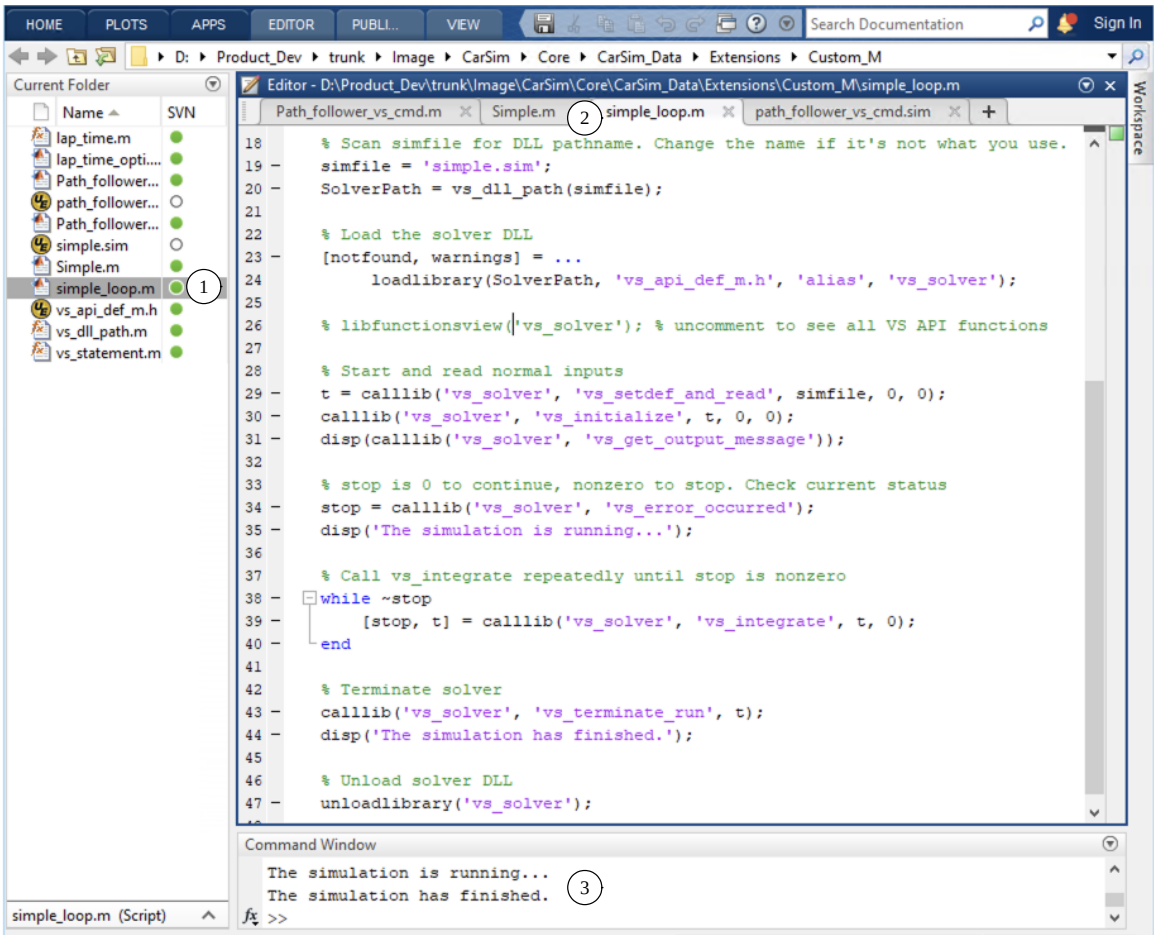


Figure 9. The simple_loop M-File has a structure suited for extending models.

The difference is that in `Simple.m`, a single command runs the simulation using the VS API function `vs_run` via the MATLAB function `calllib`, whereas in `simple_loop.m` (Figure 9) the simulation is done in several steps. After the DLL is loaded, the model is initialized with four API function calls:

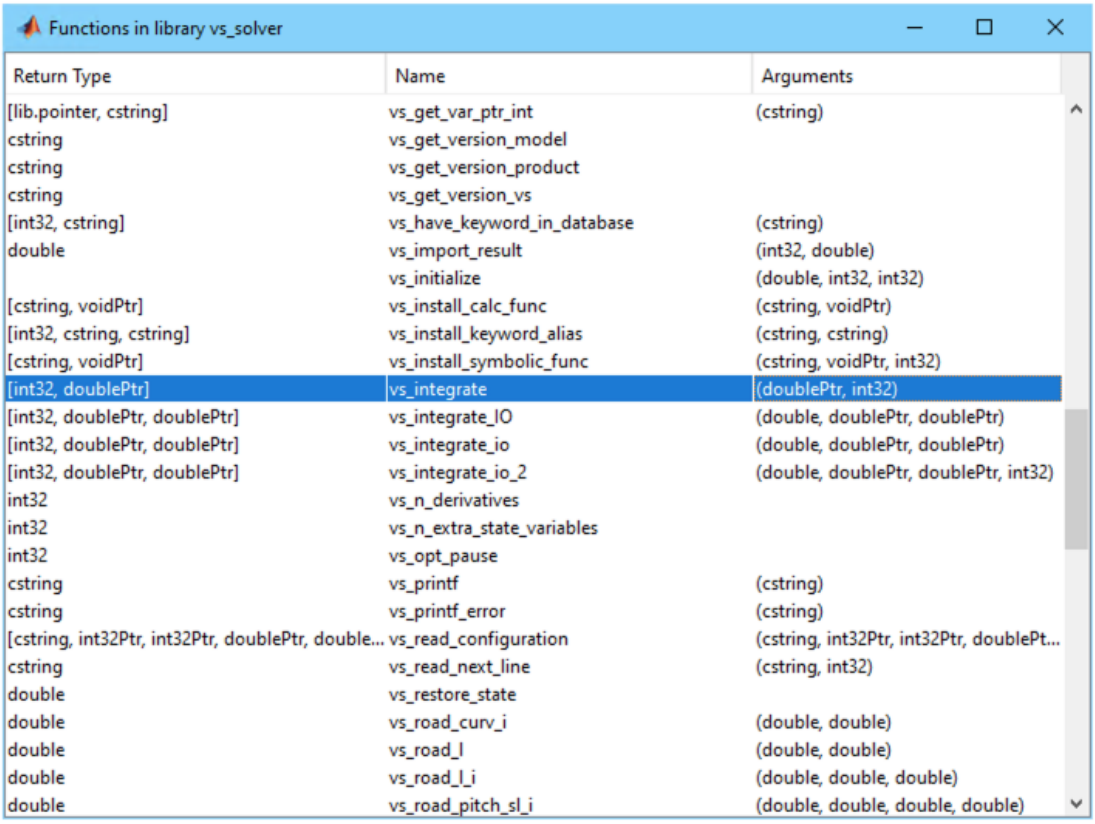
1. The API function `vs_setdef_and_read` (line 29) builds a VS Math Model and commands it to read all input Parsfiles. The function also returns the start time, which is assigned to a MATLAB variable `t`.
2. The API function `vs_initialize` (line 30) instructs the VS Math Model to calculate initial conditions based on all of the information read from the input Parsfiles
3. The API function `vs_get_output_message` (line 31) provides text that is displayed in the MATLAB Command Window (3).
4. The API function `vs_error_occurred` (line 34) is used to see if any errors occurred during the initialization. The MATLAB variable `stop` is set to zero if no error occurred.

The simulation is run with a repeated call to the `vs_integrate` function (line 39). Each time step, the function updates two variables in MATLAB: `stop` and `t`. The simulation continues as

long as `stop` remains zero. The current simulation time `t` is required as an input to `vs_integrate`. The variable `t` is also an output that is updated by `vs_integrate`.

When the simulation ends (`stop` is no longer zero), the API function `vs_terminate_run` is called to terminate the run (line 43). This cleans up memory and closes output files before the DLL is unloaded (line 47).

The `simple_loop` example includes a commented out statement using the `libfunctionview` function (line 26). If the statement is active, it causes a new window to appear (Figure 10) with all of the API functions in alphabetical order. This is not needed for normal operation but is sometimes helpful when learning about the VS API, or when debugging.



Return Type	Name	Arguments
[lib.pointer, cstring]	vs_get_var_ptr_int	(cstring)
cstring	vs_get_version_model	
cstring	vs_get_version_product	
cstring	vs_get_version_vs	
[int32, cstring]	vs_have_keyword_in_database	(cstring)
double	vs_import_result	(int32, double)
	vs_initialize	(double, int32, int32)
[cstring, voidPtr]	vs_install_calc_func	(cstring, voidPtr)
[int32, cstring, cstring]	vs_install_keyword_alias	(cstring, cstring)
[cstring, voidPtr]	vs_install_symbolic_func	(cstring, voidPtr, int32)
[int32, doublePtr]	vs_integrate	(doublePtr, int32)
[int32, doublePtr, doublePtr]	vs_integrate_IO	(double, doublePtr, doublePtr)
[int32, doublePtr, doublePtr]	vs_integrate_io	(double, doublePtr, doublePtr)
[int32, doublePtr, doublePtr]	vs_integrate_io_2	(double, doublePtr, doublePtr, int32)
int32	vs_n_derivatives	
int32	vs_n_extra_state_variables	
int32	vs_opt_pause	
cstring	vs_printf	(cstring)
cstring	vs_printf_error	(cstring)
[cstring, int32Ptr, int32Ptr, doublePtr, double...	vs_read_configuration	(cstring, int32Ptr, int32Ptr, doublePt...
cstring	vs_read_next_line	(cstring, int32)
double	vs_restore_state	
double	vs_road_curv_i	(double, double)
double	vs_road_l	(double, double)
double	vs_road_l_i	(double, double, double)
double	vs_road_pitch_sl_i	(double, double, double, double)

Figure 10. VS API functions displayed with MATLAB `libfunctionsview` function.

As noted earlier, the results obtained using `simple_loop.m` are identical to those obtained with `Simple.m`, which are in turn identical to running the VS Math Model directly from the VS Browser without any involvement of MATLAB.

The reason for showing and explaining the layout of `simple_loop.m` is that this program will be extended in two following examples that show methods for adding the path follower to the vehicle model.

Applying VS Commands from MATLAB

There are situations where automation provided with MATLAB might be combined with standard inputs for the VS Math Model. A VS API function `vs_statement` can be used to provide almost

any input to the VS Math Model that can be provided from a Parsfile input as a single line of text. The only exceptions are tables of numbers, which require multiple lines of text.

For example, the VS Commands shown earlier to add the path follower to a VS Math Model can be specified within a MATLAB file using the `vs_statement` function. Normally, VS Commands are provided in datasets managed by the VS Browser, and sent to the VS Math Model as text within Parsfiles (Figure 2, page 6). The same effect is obtained by applying the VS Commands using the `vs_statement` function from MATLAB (Figure 11).

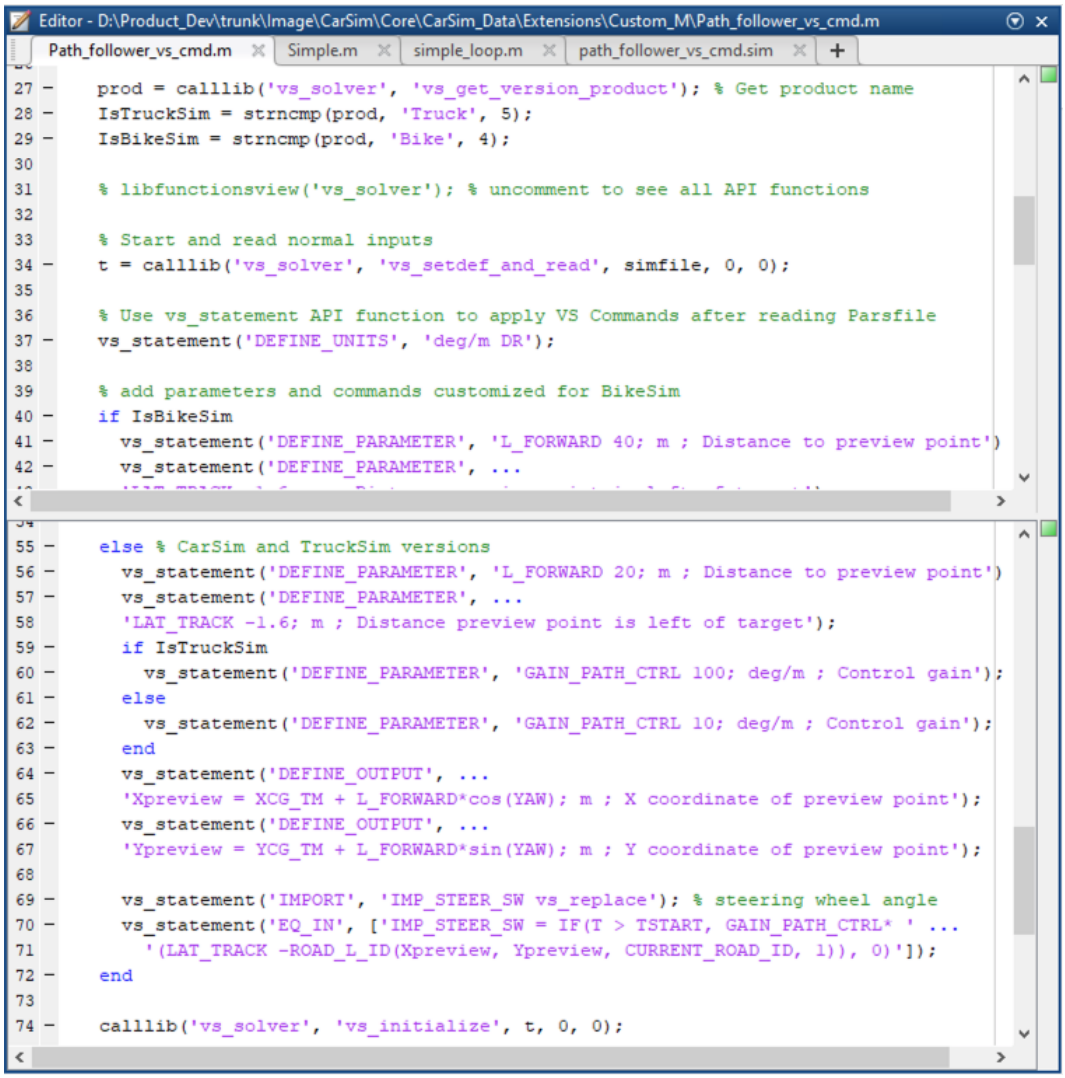


Figure 11. Adding the simple path follower with VS Commands from MATLAB.

The M file `Path_follower_vs_cmd.m` was made by modifying the M file `simple_loop.m` shown earlier (Figure 9). In comparing the two M files, you can see that the `simple_loop.m` file has a call to `vs_setdef_and_read` (line 29, Figure 9) immediately followed by a call to `vs_initialize` (line 30). In the `Path_follower_vs_cmd.m` file, a number of VS Commands were inserted in between those existing statements (lines 34 and 74, Figure 11).

Compare the display of the VS Commands at the end of the Echo file shown earlier (Figure 3, page 7) with the inputs to the vs_statement MATLAB function (lines 56 – 71, Figure 11).

This example M file works with CarSim, TruckSim, and BikeSim, using slightly different VS Commands for the different products. The VS API function vs_get_version_product is used to get a string identifying the product and version (line 27). This string, assigned to the variable prod, will begin with the text CarSim, TruckSim, or BikeSim. Two variables are set to 0 or 1 using the MATLAB function strncmp to compare the first 4 or 5 characters of the variable prod to the strings "Truck" and "Bike" (lines 28 and 29). If the product is TruckSim, then IsTruckSim = 1; if the product is BikeSim, then IsBikeSim = 1. If not, they are set to 0. If both are 0, the product is assumed to be CarSim.

These two Boolean variables are used to activate blocks of VS Commands. If IsBikeSim is not zero, then lines 41-54 are evaluated (not shown). Otherwise statements in lines 56-71 are evaluated. Lines 59-63 distinguish between CarSim and TruckSim using different default values for the parameter GAIN_PATH_CTRL for TruckSim (100 deg/m) or CarSim (10 deg/m).

When a simulation is run using the Path_follower_vs_cmd.m file, the resulting Echo file is nearly identical to the Echo file generated without MATLAB (e.g., Figure 3, page 7). Either way, the VS Math Model receives the same VS Commands. Without MATLAB, it was receiving the information from the datasets in the database (Figure 2, page 6). With MATLAB, it was receiving the information after all Parsfile information had been scanned (vs_setdef_and_read) but before the initialization was performed (vs_initialize).

Note Functions from a DLL are applied using the MATLAB calllib function (see lines 27, 34, and 74 in Figure 11). A statement using callib needs its own name (calllib), the name of the loaded DLL (vs_solver), the name of the DLL function (e.g., setdef_and_read), and additional arguments that are needed for the DLL function, as indicated in the diociumentation or the window made via the MATLAB libfunctionsview function. (Figure 10).

For this example, a MATLAB M-file function was made named vs_statement that in turn calls the vs_statement function from the DLL, with just two arguments (keyword and statement string). This was done for the purpose making the code shorter and more readable, given the multiple VS Commands.

Using Import and Export Arrays with MATLAB

When running with Simulink, information is shared between the Simulink model and the VS Math Model using arrays of Import and Export variables. These arrays are always available for use in VS Math Models for sharing information with external programs, and may be used with MATLAB, even without Simulink. Using the Import and Export arrays, the equations used for the example steering controller can be applied in MATLAB. To do this:

1. Arrays must be defined in MATLAB whose contents are the variables of interest.

2. Import and Export variables must be activated in the VS Math Model to match the definitions from MATLAB.
3. Every time step, values for the array elements imported to the VS Math Model must be calculated in MATLAB, and the contents of the arrays must be shared between MATLAB and the VS Math Model.

The approach shown in the previous subsection for using VS Commands is again used in the example file `Path_follower.m`. Figure 12 shows that some VS Commands are again applied after the call to `vs_setdef_and_read` (line 35) and before the call to `vs_initialize` (line 63). As in the previous example, variables `IsBikeSim` and `IsTruckSim` are created allow the same M file to work with CarSim, TruckSim, and BikeSim.

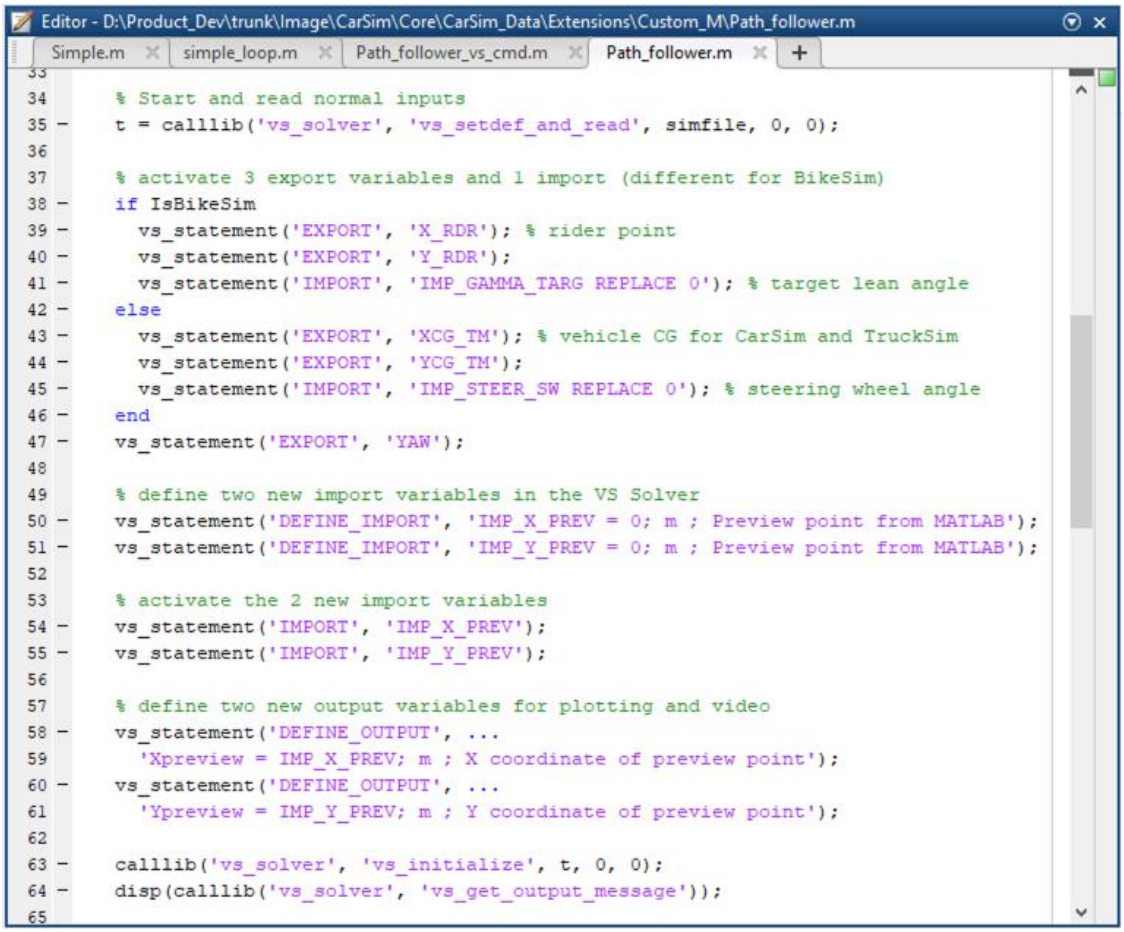


Figure 12. Setting up Import, Export, and Output variables for the VS Math Model.

The vehicle Yaw and X and Y coordinates are needed to calculate a control angle (steer or lean) in MATLAB and must therefore be exported. These variables are activated for Export with three `EXPORT` VS Commands. Two of the variables are different for BikeSim (lines 39 and 40) and the others (lines 43 and 44).

The control to be imported also differs for BikeSim (line 41) and the others (line 45).

For the purpose of this example, the calculations of the global X and Y coordinates of the preview point are performed in MATLAB. In order to use these values in VS Visualizer for plotting and animation, they must be imported from MATLAB, and assigned to output variables in the VS Math Model.

To do this, two new Import variables (`IMP_X_PREV` and `IMP_Y_PREV`) are defined with the `DEFINE_IMPORT` VS Command (lines 50 and 51). These are activated for Import with the `IMPORT` command (lines 54 and 55).

Two new output variables are defined with the same names as before (`Xpreview` and `Ypreview`, lines 58 - 61). However, in this case, the formula for each is simply the symbol for the corresponding Import variable (`IMP_X_PREV` and `IMP_Y_PREV`, respectively).

To summarize: the code shown in Figure 12 sets up the VS Math Model to export three variables each time step (X and Y coordinates of a point in the model, and `YAW`) and to import three variables each time step (`IMP_X_PREV`, `IMP_Y_PREV`, and a control that is lean for BikeSim and Steer for the others).

Figure 13 shows the continuation of `Path_follower.m`. In this example, the parameters used to calculate steering wheel angle are MATLAB variables that are assigned values (lines 67 – 70). Alternative values are set for BikeSim (lines 74 and 75) and TruckSim (line 77). The remaining lines of code apply for all three products.

In order to match the setup of the VS Math Model for an Import array with three variables and an Export array with three variables, two arrays are defined in MATLAB with three elements each (lines 81 and 82). Arrays are accessed with VS API functions using pointers, so a pointer to each array is defined (lines 83 and 84).

As in the previous examples, a `while` loop is used to perform calculations at each time step (lines 93 - 113). In the other example, the loop contained a single MATLAB statement using the VS API function `vs_integrate` (line 39, Figure 9, page 12). However, in this case, the loop contains more statements, needed to make use of variables exported from the VS Math Model to calculate the three variables that will be imported to the VS Math Model. In support of the transfer of variables using arrays, a different API function is used (`vs_integrate_io`, line 112), with additional arguments.

Each time step, two actions are needed in MATLAB to connect the values within the arrays to the pointer variables used to communicate with the VS Math Model.

1. Values from the VS Math Model are copied to MATLAB using the `get` function (line 97).
2. The pointer sent to the VS Math Model for the Import array is updated each time step using the `set` function (line 109).

In addition to the two pointers, the API function `vs_integrate_io` needs the current simulation time `t`. A statement at the top of the loop (line 94) updates `t` using the time step, which was obtained from the VS Math Model after the initialization using the API function `vs_get_tstep` (line 87).

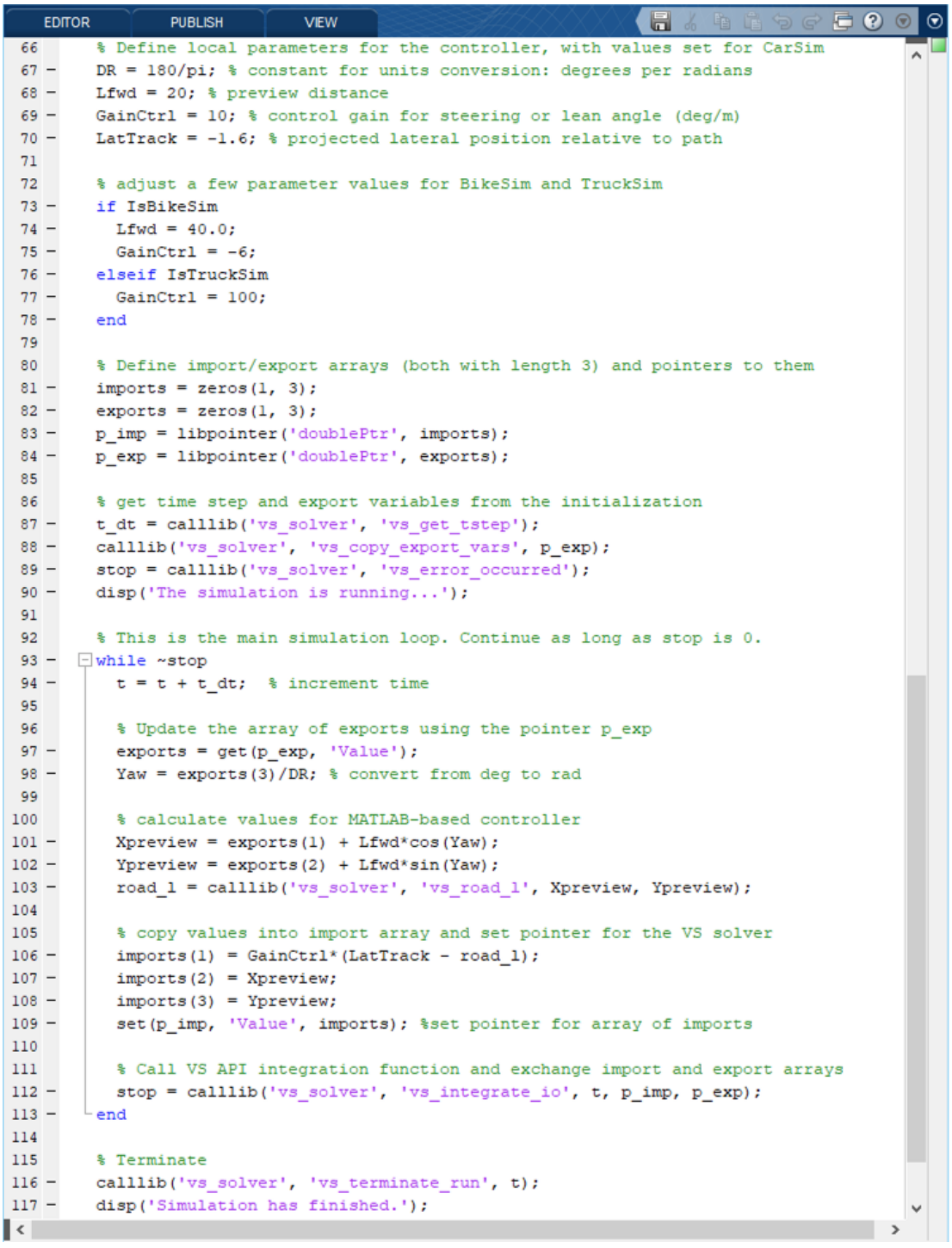


Figure 13. Using arrays of Import and Export variables within MATLAB.

Each time through the loop, MATLAB needs the values of the three output variables exported by the VS Math Model. Before starting the loop, the pointer to the Export array from the VS Math Model is updated using the API function `vs_copy_export_vars` (line 88). Every time step, the pointer is updated by `vs_integrate_io` (line 112).

The X and Y coordinates for the preview point are calculated using the exported X and Y coordinates and vehicle yaw (lines 101 and 102). The lateral position of the preview point is calculated with the VS API equivalent of the `ROAD_L` function (line 103) and used to determine, the control angle (steer or lean), which is the first value in the Import array (line 90). In addition, the X and Y coordinates of the preview point are sent to the VS Math Model using the second and third elements of the Import array (lines 107 and 108).

As with the other examples, the function `vs_terminate_run` is called (line 116) when the run finishes, after which the DLL is unloaded.

This example is run using the same type of setup as the other MATLAB examples.

When comparing the results from this example with results obtained using VS Commands without MATLAB or VS Commands from MATLAB (as described in the previous subsection), identical results are obtained in terms of plots and videos. However, the Echo file obtained using Import and Export arrays differs (Figure 14). In comparing this Echo file to the Echo file shown earlier for the extension done solely with VS Commands (Figure 3, page 7), note the following differences:

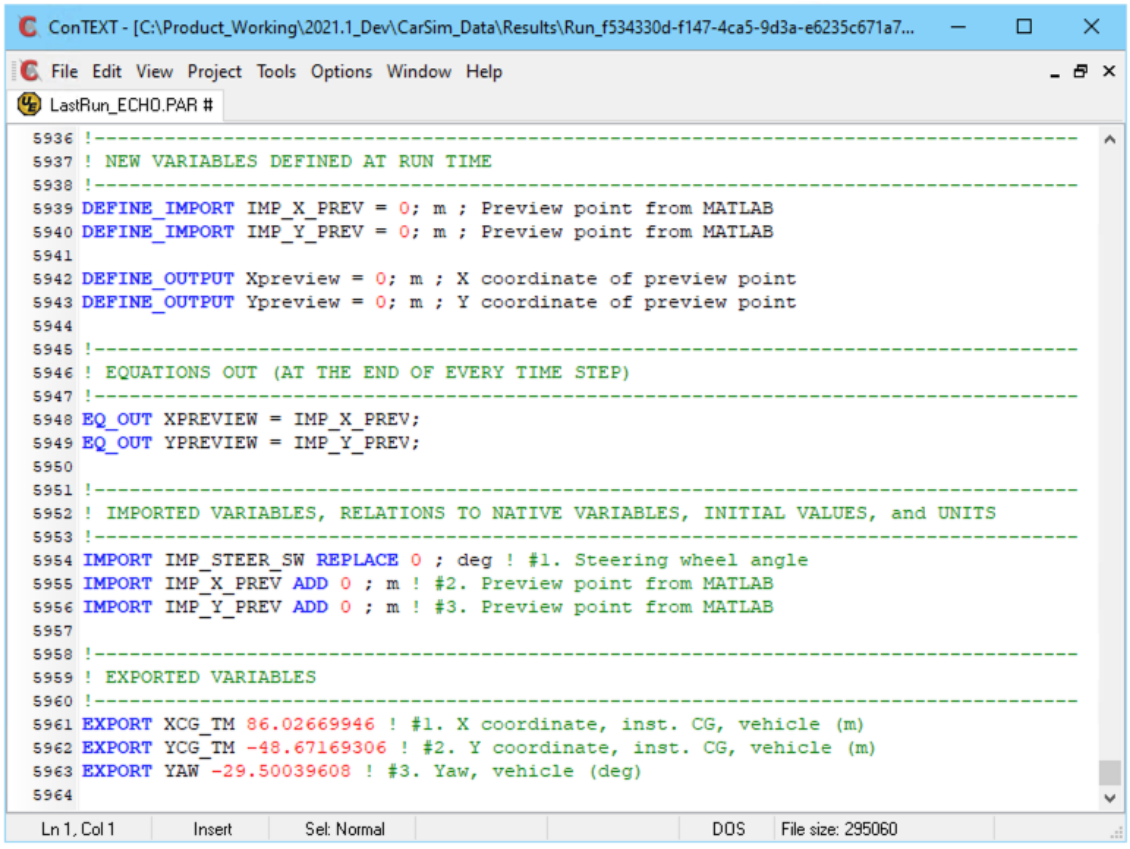


Figure 14. Echo file for MATLAB example using Import and Export arrays.

1. The VS Command version defines three new parameters (`L_FORWARD`, `LAT_TRACK`, and `GAIN_STEER_CTRL`) that are not echoed by the VS Math Model for the array version. (They are variables defined within MATLAB.)

2. Two new IMPORT variables (IMP_X_PREV and IMP_Y_PREV) are defined in the array version.
3. The VS Command version includes an EQ_IN equation for IMP_STEER_SW that is not part of the array version.
4. The EQ_OUT commands used to calculate the output variables Xpreview and Ypreview have full formulas in the VS Command version, but simply copy the values of Import variables in the array version.
5. The array version includes two IMPORT and three EXPORT commands that do not exist in the VS Command version.

On the other hand, both methods activate the IMP_STEER_SW Import variable (in BikeSim, the import is IMP_GAMMA_TARG), and both define two new output variables for the global coordinates of the preview point.

Extending a Model with C/C++

Methods for extending VS Math Models used in high-level languages such as MATLAB can be also used in C. However, when running a VS Math Model from a wrapper program written in C/C++, several options are available that are not possible with MATLAB, because the wrapper can use C pointers to variables and functions:

1. The wrapper program can directly access and possibly change any variable within the VS Math Model that has an identifying keyword; Import and Export arrays are not needed.
2. The wrapper program can install I/O support for new variables that exist in the wrapper, but which may be handled by the VS Math Model as parameters, output variables, state variables, etc.
3. The wrapper program can provide callback functions that are applied automatically by the VS Math Model at specified times within the simulation process. The callback functions can in turn access existing variables, add new variables, perform calculations, etc.

Review: Running a C Wrapper EXE Program

As described in the tech memo *Automating Runs with the VS API*, the **Run Control** dataset for a simulation made with a wrapper EXE should include a link (2) (Figure 15) to a dataset in the library **Models: Self-Contained Solvers** (4) selected with the **Models** drop-down control (3). With this setup, you can run a simulation by clicking the **Run Math Model** button (1), just as with the built-in VS Math Models.

In the linked **Models: Self-Contained Solvers** dataset (Figure 16), check the box to use an external wrapper program (1) and specify the pathname for the compiled wrapper program (2).

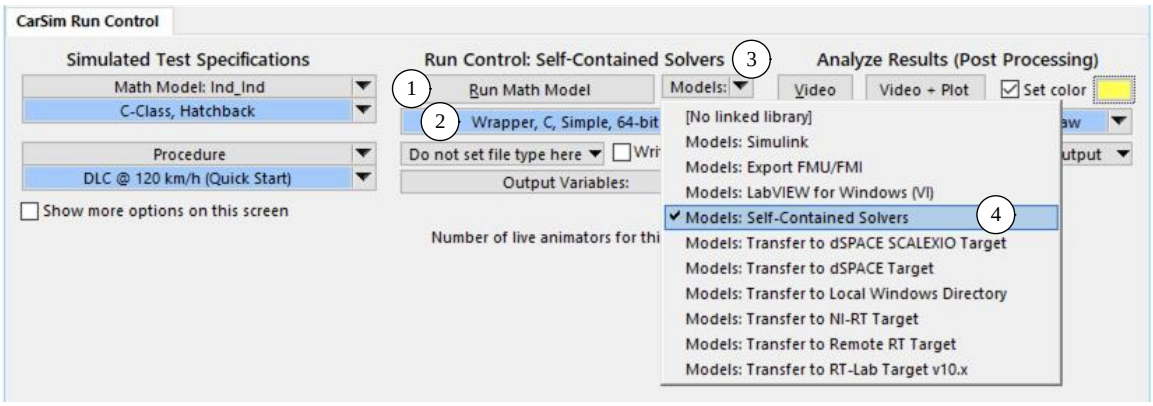


Figure 15. Link to the library Models: Self-Contained Solvers.

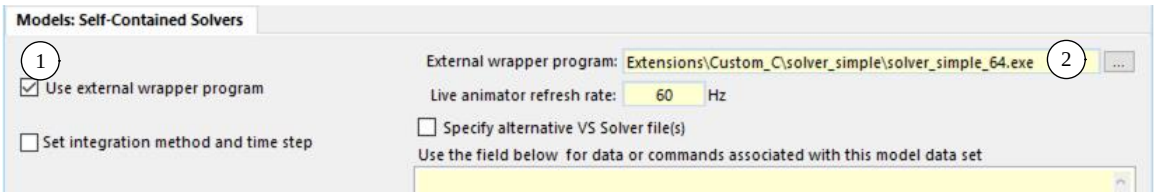


Figure 16. Identify the pathname to the compiled EXE wrapper program.

Applying VS Commands from a C Wrapper Program

The path follower was implemented in two ways in the earlier MATLAB examples, and the same two methods will be demonstrated for C.

The first involved applying the VS Commands from MATLAB. A similar example shows how to apply VS Commands from a C wrapper program, contained in the folder `Extensions\Custom_C\wrapper_vs_cmd_path`.

Figure 17 shows the source code that handles some of the initialization part of the simulation. As seen in other examples, it uses a text string `simfile` to identify the Simfile (lines 34 and 38), uses a function `vs_get_dll_path` to scan the Simfile and determine the pathname for the VS Solver DLL (line 39), loads the VS Solver DLL (line 41), gets the VS API functions (line 47), and uses the API function `vs_setdef_and_read` to start the VS Math Model and have it read input Parsfiles (line 55).

This example also uses two local variables to handle differences for TruckSim (line 51) and BikeSim (line 52).

The API function `vs_statement` is used to provide the same VS Commands that were shown earlier for a dataset managed by the VS Browser (Figure 2, page 6) and as specified in a MATLAB M file (Figure 11, page 14). The same statements are used in this C example (lines 63 – 75 show VS Commands for BikeSim; commands for CarSim and TruckSim start on line 78). When the wrapper program is run, the same results are obtained as were seen before in the simulation results and the appearance of the VS Commands at the end of the Echo file.

Using Callback Functions from a C Wrapper Program

The example path follower is added to the model using another C wrapper program from the folder `Extensions\Custom_C\wrapper_path_follower`. In this case, the model extension is defined using C callback functions. The parameters for the example path follower will not only be used in externally programmed equations, but they will be installed into the VS Math Model so new values can be set at runtime using the same VS GUI and database. In addition, the values for the parameters will be shown in the Echo files generated with each run.

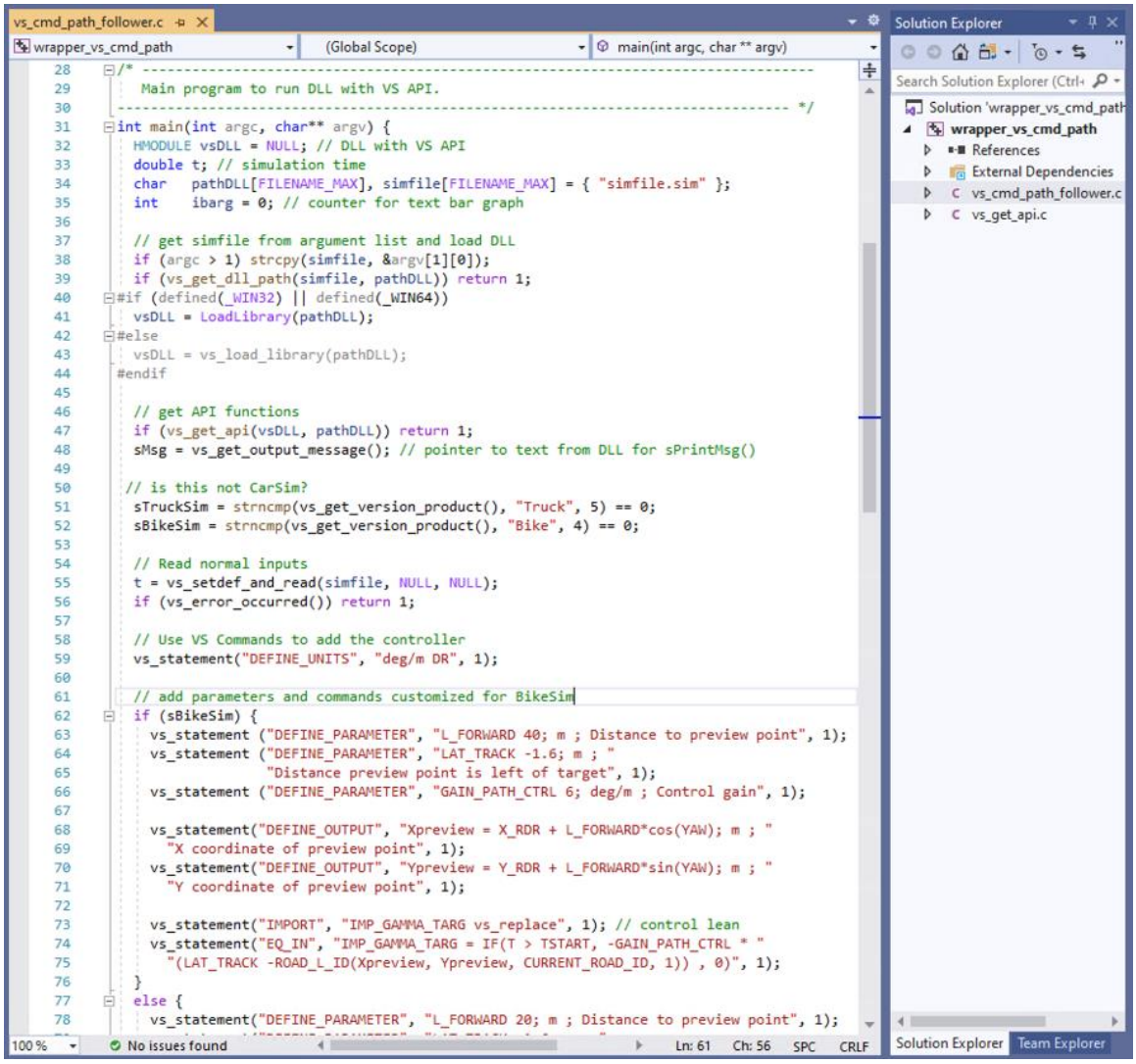


Figure 17. C Wrapper Program that applies VS Commands.

Figure 18 shows C code for `wrapper_extended.c`. This is much simpler than the previous example; it is similar to the program described in the tech memo *Automating Runs with the VS API*. It also scans a specified Simfile and loads the DLL for the VS Solver (lines 28 and 30), gets the API functions from the DLL (line 35), and uses the API function `vs_run` to make the simulation and show a popup message if something goes wrong (lines 44 and 45). After handling a possible pause (lines 51 – 56), the DLL is freed and the program exits.

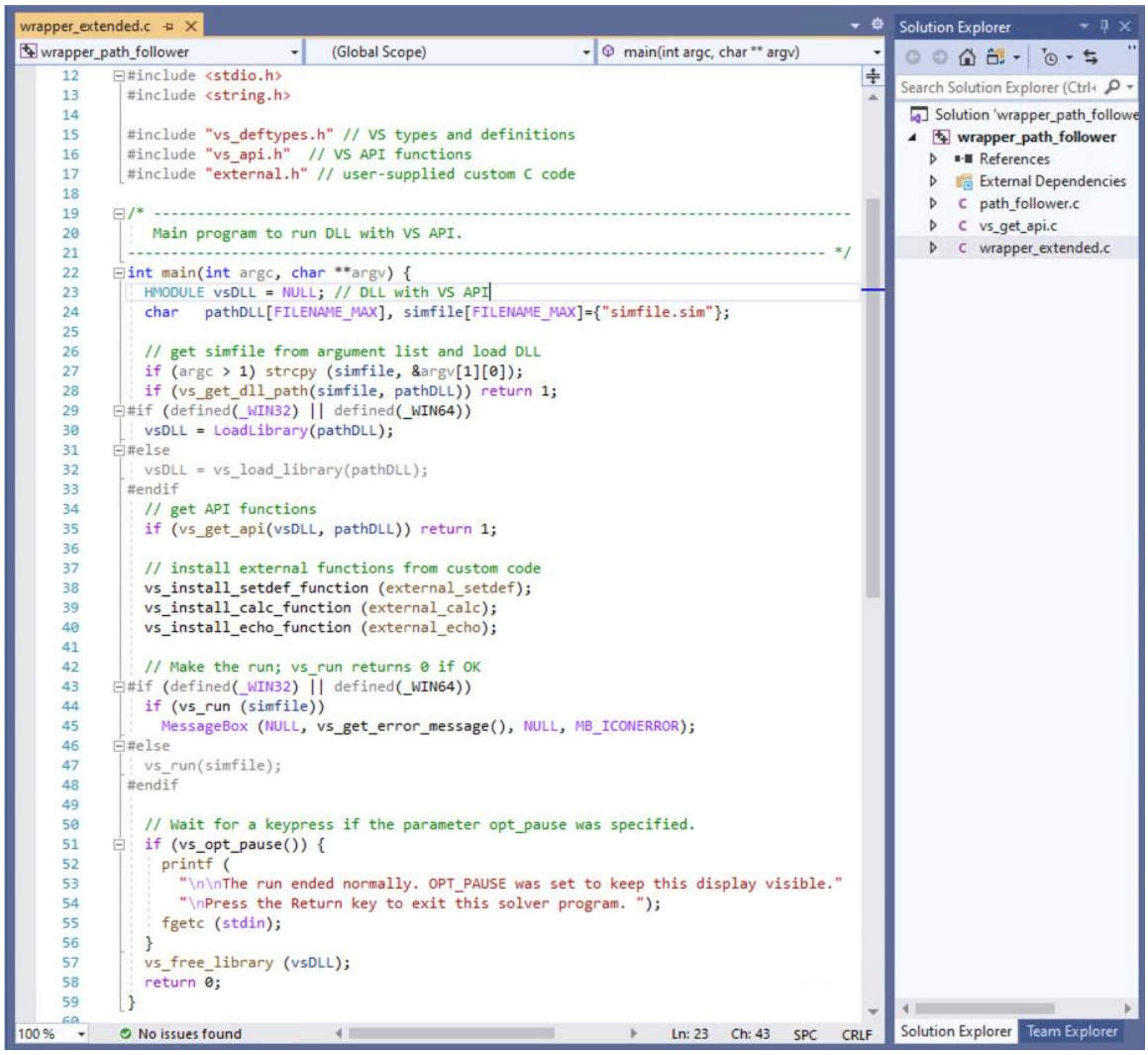


Figure 18. Main program for C wrapper that uses callback functions.

However, unlike the simpler automation example, this wrapper installs externally defined functions that are deployed at various times when the VS Math Model runs (lines 38 – 40).

The file shown (`wrapper_extended.c`) is generic; the details of the model extension are provided by the three functions: `external_setdef`, `external_calc`, and `external_echo`. These are defined in a separate file named `path_follower.c`.

In many cases, including the path follower example, only two of these functions are needed to add new model features: `external_setdef` and `external_calc`. The other handles custom cases for writing information into the Echo file.

Figure 19 shows the top part of `path_follower.c`, with the `#include` statements and declarations of static variables used to support the example path follower (lines 22 - 30). Most of the variables are declared with type `vs_real`, which is normally set to the C type `double` in the header file `vs_deftypes.h`. There are also three integer variables: `sUserExternal` `sBikeSim`, and `sTruckSim` (lines 29 and 30).

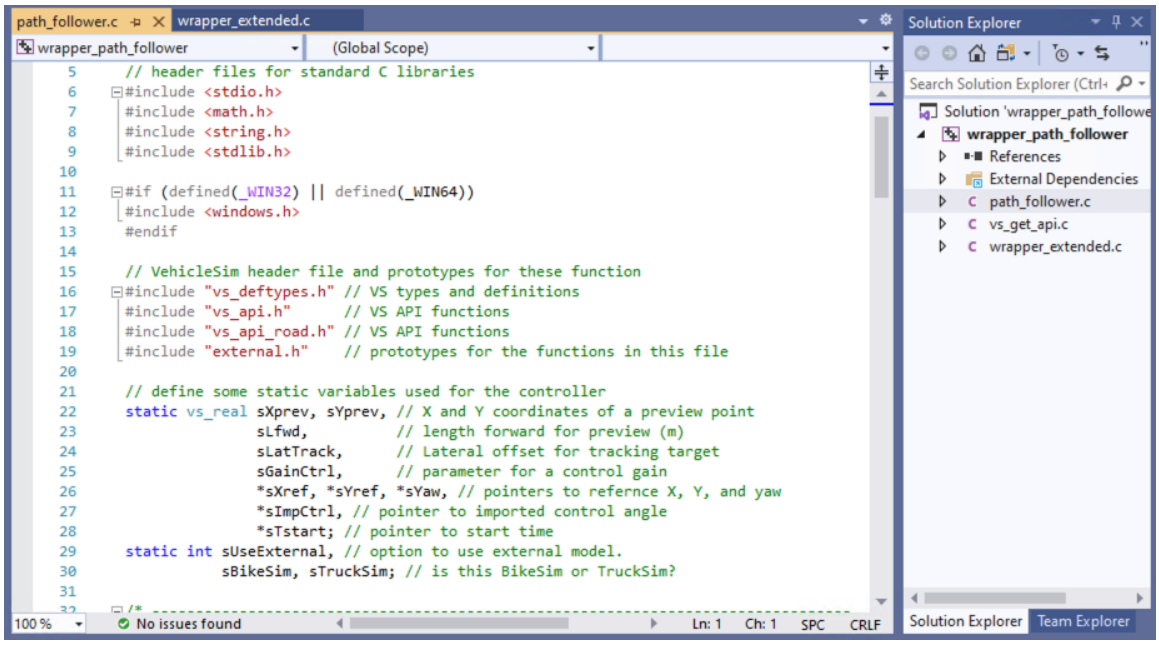


Figure 19. Required #include statements and shared static variables.

Setting up the external code with external_setdef

Figure 20 shows part of the source code for `external_setdef`, the function that defines new variables and links to the variables in the VS Math Model. This is the name that was specified earlier using the API function `vs_install_setdef_function` (line 38, Figure 18).

The function `external_setdef` provides the same information that has been used in other examples:

- Lines 43 and 44 assign Boolean values to the variables sTruckSim and sBikeSim.
- Line 47 (Figure 20) has the API equivalent of the VS Command `DEFINE_UNITS`, and installs the "deg/m" units for the steering control gain.
- Lines 55 - 60 define the three parameters used in the VS Command examples: `L_FORWARD`, `LAT_TRACK`, and `GAIN_STEER_CTRL`. Here, the API function `vs_define_par` is used to specify all properties of the parameter that were set before (keyword, default value, keyword for units, description), along with properties that cannot be set with VS Commands (e.g., pointer to an existing variable used for the parameter).
- The variables `sTruckSim` and `sBikeSim` are used to provide alternative arguments using the inline C form *cond ? opt1 : opt2*. For example, in line 55 the default value for BikeSim is 40; otherwise it is 20. Line 59 shows different values being set for BikeSim (6), TruckSim (100), and CarSim (10). Lines 60 and 61 show alternative descriptions for the parameter `GAIN_PATH_CTRL`.

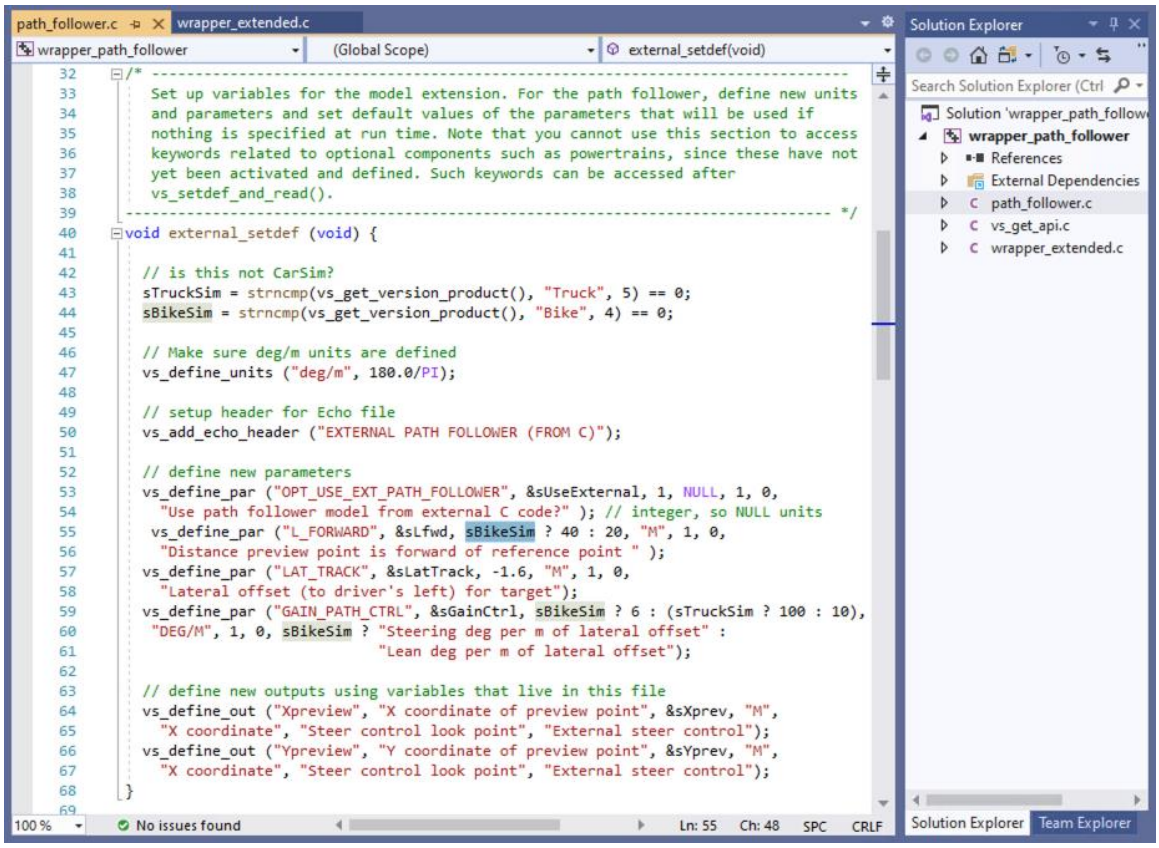


Figure 20. Source code for the function external_setdef.

- Lines 64 - 67 define the two output variables added with the VS Command examples: `Xpreview` and `Ypreview`. Here, the API function `vs_define_out` is used. Again, the API function sets all of the properties that were set in the VS Commands (keyword, long name, keyword for units, component name), along with a pointer to the variable used for the output and two additional labels (generic name and category name).

This example adds a few more features not available from VS Commands. A heading is set up for the Echo file (line 50), and the static variable `sUseExternal` is used as an integer parameter to activate or disable the external steering model (lines 53 and 54).

Parameters defined with the API function `vs_define_par` are listed in the Echo file in the order in which they are introduced. Thus, the four parameters defined here (lines 53 – 61) will be shown in the Echo file together, immediately after the header that was installed with the `vs_add_header` function (line 50). For example, Figure 21 shows part of the Echo file generated when using this custom C program.

To summarize: all of the parameters and output variables used for this example are installed as native parameters and outputs. Links have been made to five existing variables within the VS Math Model. The only remaining code that is needed is to perform calculations for the global coordinates of the preview point and the value of the imported control angle (steer or lean).

```
82 TSTOP              38 ; s ! Stop when this time is reached
83 ! T_DT         0.0005 ; s ! CALC -- Time increment between calculations
84
85 !-------------------------------------------------------------------------------
86 ! EXTERNAL STEERING MODEL (FROM C)
87 !-------------------------------------------------------------------------------
88 OPT_USE_EXT_STEER   1 ! [D] Use steering model from external C code?
89 L_FORWARD          20 ; m ! [D] Distance preview point is forward of vehicle CG
90 LAT_TRACK        -1.6 ; m ! [D] Lateral offset (to driver's left) for target
91 GAIN_STEER_CTRL    10 ; deg/m ! [D] Control parameter: steering angle per meter of
92                       ! lateral offset
93
94 !-------------------------------------------------------------------------------
95 ! SYSTEM CONSTANTS
96 !-------------------------------------------------------------------------------
97 ! DR      57.29577951 ; - ! Deg/rad symbol to use in formulas (read-only)
98 ! G           9.80665 ; - ! Symbol for gravity constant m/s/s (read-only)
```

Figure 21. Display of new parameters added to VS model.

Performing calculations with external_calc

Figure 22 shows the source code for `external_calc`, the callback function that calculates the steering wheel angle and the coordinates of the previous point. This function has two arguments: the current simulation time and an integer macro that identifies the location in the equations. As noted in the comments (and in the *VS API Manual*), this function can be used in nine places. In this example, only three are used:

1. `VS_EXT_AFTER_READ` (after the input Parsfiles have been read but before the initialization is performed),
2. `VS_EXT_EQ_IN` (the beginning of each time step, when calculated values will affect other variables in the model), and
3. `VS_EXT_EQ_OUT` (near the end of each time step, when most other variables in the model have already been calculated).

Some of the existing variables in the VS Math Model are used, and pointers are obtained for them using the API function `vs_get_var_ptr` after the Parsfile imports have been read, but before any calculations are made with the new path follower (lines 77 – 90). Again, the variable `sBikeSim` is used to include code for getting pointers to variables that differ in BikeSim and the others (e.g., `X_RDR` in BikeSim or `XCG_TM` in the others).

Because an existing Import variable (`IMP_GAMMA_TARG` for BikeSim or `IMP_STEER_SW` for the others) is used to apply the calculated steering wheel angle, it is automatically activated with the `IMPORT` VS Command after all Parsfiles have been read (`VS_EXT_AFTER_READ`) if the new parameter `sUseExternal` is not zero (lines 81 and 87).

As was the case for the implementation with VS Commands, the import control angle is calculated at the beginning of each time step (`VS_EXT_EQ_IN`, lines 95 - 97) because the model response is strongly affected by this control. The sign is changed if this is BikeSim (line 98).

The two coordinates of the preview point are calculated at the end of each time step (`VS_EXT_EQ_OUT`, lines 103 and 104) because they depend on the location of the vehicle, which has not been updated at the start of the time step.

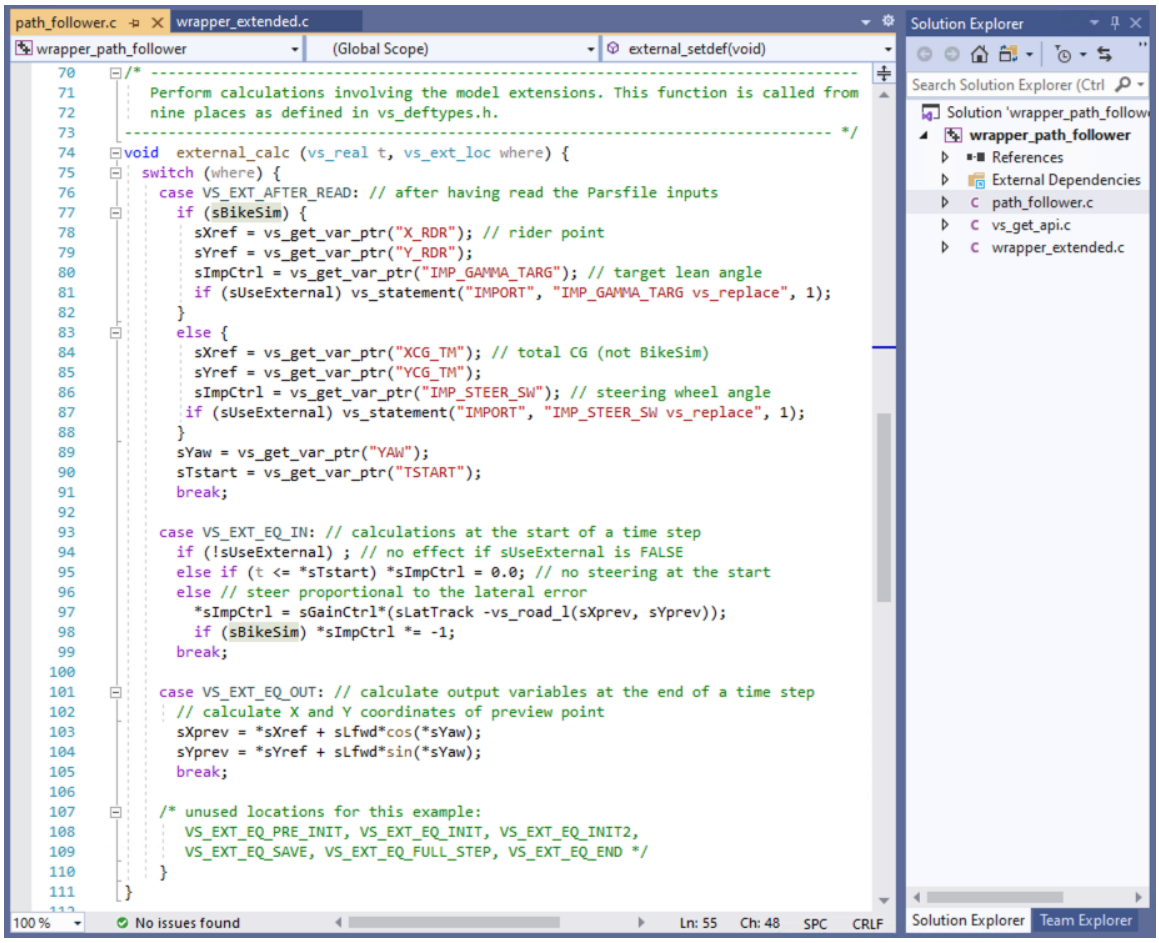


Figure 22. Source code for the function external_calc.

The two coordinates of the preview points are always calculated, regardless of the state of `sUseExternal`.

Customizing the Echo file

Most extensions to VS Math Models will require only the two functions that are installed with function `vs_install_setdef_function` (set up shared variables) and function `vs_install_calc_function` (make calculations).

However, the API also supports extensions that might use non-standard input formats or writing into the Echo files. The example shows how the Echo file can be extended to show additional information using custom code, in a function `external_echo` (Figure 23). When the VS Math Model writes an Echo file, it will call this function in four places, identified with the argument `where`. A VS API function `vs_write_to_echo_file` is used to send a string to the DLL so the VS Math Model can in turn write the string to file.

The code in this example writes additional text at the top of the Echo file to tell users that the run was made with an extended model (lines 125 - 127). This generates information in the Echo file seen in Figure 24 (lines 6 and 7).

As shown earlier, the Echo files generated with the extended model show the new parameters just as if they were built in, with the keyword, value, units, and description (Figure 21, page 26). However, be aware that new output variables are not listed in text files and spreadsheets accessed with the **View** button the **Run Control** screen, or when browsing to select output files for a **Plot Setup** dataset. This is because the VS Browser obtains the documentation files with direct calls to the DLL without the effect of any wrapper programs.

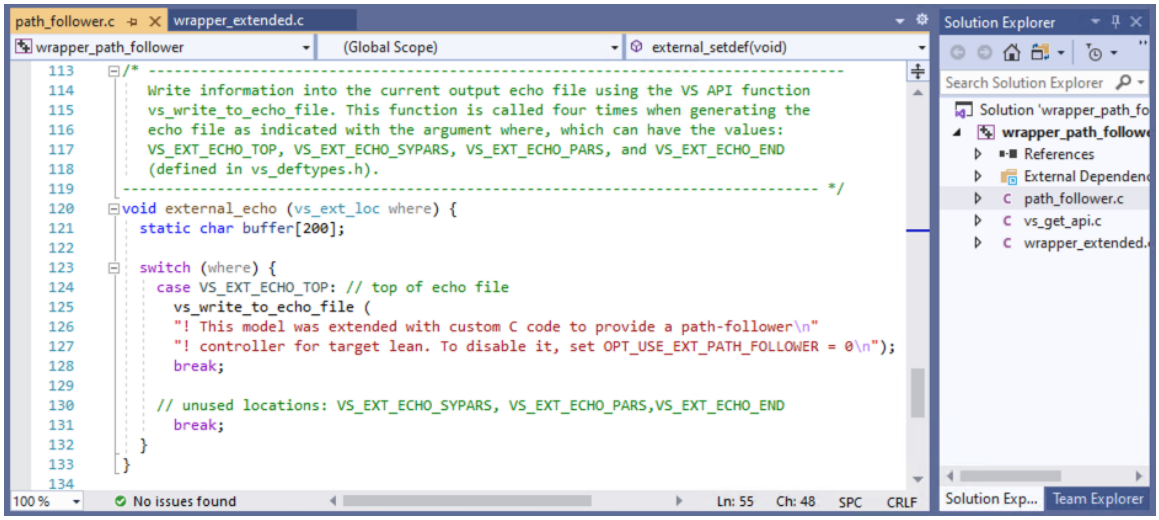


Figure 23. Example function external_echo.

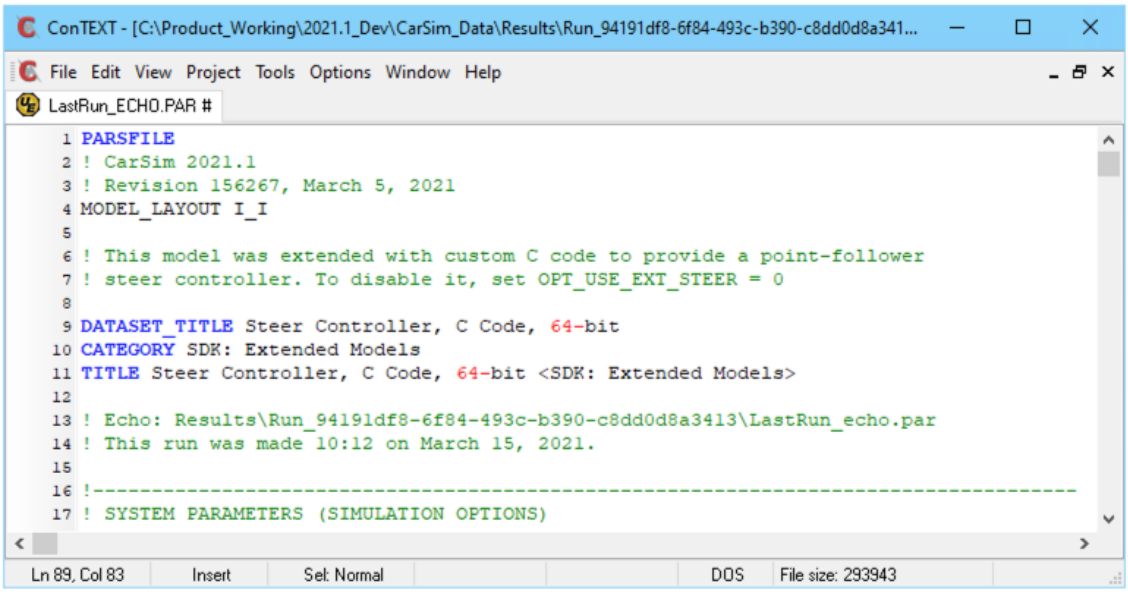


Figure 24. Top of Echo file from example C wrapper program.

Summary

This document shows how a VS Math Model can be extended using several methods. The example is a path follower, in which a CarSim or TruckSim vehicle is steered in proportion to the lateral

distance between a preview point a fixed distance in front of the vehicle, and the target path (a constant distance to the left or right of a road centerline). In BikeSim, the controller is used to calculate target bike lean.

The first approach uses VS Commands and requires no additional software beyond CarSim, TruckSim, or BikeSim (other than a C compiler). To extend the model with VS Commands:

1. Read through the detailed documentation (readme text files and Echo files) to find the existing Import and output variables of interest. Also, review the Echo file for any existing parameters that would be used in equations you will add to the model.
2. Add variables to the model using VS Commands such as `DEFINE_PARAMETER`, `DEFINE_OUTPUT`, and others (see VS Commands).
3. Add equations to the model using VS Commands such as `EQ_IN`, `EQ_OUT`, `EQ_DIFFERENTIAL`, and others (see VS Commands).
4. Add extra information for user-friendliness, such as new units (e.g., "deg/m") and labels for output variables.

Other methods were shown that apply the VS API to build a VS Math Model under the control of another program. External programs can load the VS Solver DLL and apply input statements, including VS Commands, programmatically rather than by writing them to Parsfiles that were read by the VS Math Model. This capability was demonstrated for external programs written in MATLAB and C.

External programs can communicate with the VS Math Model each time step using arrays of Import and Export variables, as is done with Simulink and LabVIEW. This was shown for the example steering controller using MATLAB.

More options exist when the programming language is C or a language that uses pointers as they are defined in C. The external program can be given direct access to internal variables (integer and double-precision) in the VS Math Model; new variables that are physically located in the external program can be installed in the VS Math Model for seamless integration with the built-in capabilities for reading Parsfiles, writing Echo files, and processing VS Commands at runtime. An example C wrapper program was described that uses callback functions that are installed in the VS Math Model at runtime, such that they are applied automatically when needed during the simulation to provide the example path follower.